THE UNIVERSITY *of* EDINBURGH

Evolution Through Reputation: Noise-resistant Selection in Evolutionary Multi-agent Systems

Nikolaos Chatzinikolaou

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2012

Abstract

Little attention has been paid, in depth, to the relationship between fitness evaluation in evolutionary algorithms and reputation mechanisms in multi-agent systems, but if these could be related it opens the way for implementation of distributed evolutionary systems via multi-agent architectures. Our investigation concentrates on the effectiveness with which social selection, in the form of reputation, can replace direct fitness observation as the selection bias in an evolutionary multi-agent system. We do this in two stages: In the first, we implement a peer-to-peer, adaptive Genetic Algorithm (GA), in which agents act as individual GAs that, in turn, evolve dynamically themselves in real-time, using the traditional evolutionary operators of fitness-based selection, crossover and mutation. In the second stage, we replace the fitness-based selection operator with a reputation-based one, in which agents choose their mates based on the collective past experiences of themselves and their peers. Our investigation shows that this simple model of distributed reputation can be successful as the evolutionary drive in such a system, exhibiting practically identical performance and scalability to direct fitness observation. Further, we discuss the effect of noise (in the form of "defective" agents) in both models. We show that the reputation-based model is significantly better at identifying the defective agents, thus showing an increased level of resistance to noise.

Acknowledgements

I would like to express my gratitude first of all to my supervisor, Dave Robertson, for his constant support throughout the course of this project. His ideas, suggestions and prompt, insightful feedback during every stage of my PhD studies were as instrumental in making this thesis a reality as was the freedom that he allowed me in my research.

Many friends and colleagues have contributed to this thesis, whether knowingly or otherwise. Special mention must be made of Paolo Besana, for his invaluable help on all things LCC, and - of course - B.B., for the inspiration.

Many thanks also are due to my family. Without their boundless love and support, I wouldn't even be here to begin with.

Last but not least, I am grateful to the EPSRC for funding this research.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Nikolaos Chatzinikolaou*)

No man is so foolish but he may sometimes give another good counsel, and no man so wise that he may not easily err if he takes no other counsel than his own. He that is taught only by himself has a fool for a master.

– Hunter S Thompson

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Foreword

1.1.1 Evolution and Natural Selection

Evolution is old news. Arguably, some of the oldest. Ever since Darwin's seminal work [Darwin, 1872] was published about one and a half centuries ago, and after the initial controversy and friction with the institutional church was reduced to a grudge and finally an inevitable, if uneasy, coexistence, a lot of people have read, thought and written about it.

Artificial Intelligence (AI), that most ambitious area of computer science, set on its intent to replicate natural intelligence - initially at human level and then at progressively more modest levels of mammals and insects, was not unaffected. And understandably so, since - if Darwin is right - the marvel that the human mind is, is nothing more than the logical outcome of many, many generations of sex.

Many scientists in the area have shifted their attempts from trying to replicate intelligence to trying to replicate the process that led to it, namely, evolution. A search for the keyword "evolution" in *CiteSeerX* [1] yields about 150,000 results, 10% of a total of 1,500,000 articles.

Of course, not all of these articles deal directly with evolution in the Darwinian sense. It would be safe, however, to say that a fair percentage of them relate to some degree to the concept of Evolutionary Algorithms (EA), which is an umbrella term used to describe the AI technique of "breeding" artificial systems using the Darwinian principle of the survival of the fittest.

[1] http://citeseer.ist.psu.edu/index

Evolutionary AI techniques have their downsides, too. One of them is their unpredictability, which stems from their stochastic nature and the reduced level of control that the designer has on the process as well as its outcome. Another downside is time. After all, it took millions of generations for the mammalian brain to evolve to the state it is in today. Despite any benefits that EAs may have, it is not always practical to have to wait for a few million years in order to come up with, say, an autonomous robot with the intellectual capacity of an amoeba. To that end, we must often take shortcuts.

Arguably, all EAs implemented as part of AI research take some form of shortcut or another. Take, for instance, the evolution of Artificial Neural Networks (ANN). An EA may be used to evolve the synaptic weights in the ANN model, and - in more extreme cases - even the topology. It would be futile from a practical perspective, however, to depend on that same process to come up with the software that implements the model, the computer that hosts it, or the electrical grid that feeds it.

1.1.2 Other Forms of Selection

Another theme of Darwin's work, albeit one not as widely publicised as the "survival of the fittest" motif, regards sexual selection [Darwin, 1870]. This theme deals with the drives that natural organisms have towards selecting partners for reproduction. And it is this aspect of evolution that provided the original inspiration for our research.

The distinction between these two types of selection lies in the fact that, when it comes to selecting a mate, any perceived measure for "fitness" can be deceiving. To put it in Miller's words [Miller, 2001],

> *Fitness is like money in a secret Swiss bank account. You may know how much you have, but nobody else can find out directly. If they ask the bank, the bank will not tell them. If they ask you, you might lie. If they are willing to mate with you if your capital exceeds a certain figure, you may be especially tempted to lie. This is what makes mate choice difficult.*

This thesis deals with artificial "organisms", or *agents*, that reside and interact in a social context. These agents may be of diverse origins, and hence must adhere to some pre-determined social norms in order to be able to interact meaningfully with each other, as well as to coordinate in order to do something useful (for us) as a system. This is where the notions of *reputation* and *social selection* come in.

1.1.3 Enter Reputation

Reputation is not as old news as evolution is - at least, not from an AI point of view. In fact, the last few years have seen an influx of research work done in that area, a fact which stems from the increasing popularity and availability of cheap computational resources, the internet, and distributed systems - in that order. In fact, reputation is now considered a mainstream focal point in Multi-agent Systems (MAS) research, the area of AI that aims in developing intelligent systems as distributed societies of agents rather than isolated entities.

Reputation mechanisms in an AI context typically deal with open distributed systems wherein there is a chance that one or more of the participating peers may deviate from common social norms either in error, out of malice, or by an incentive to maximise their own benefits without a regard of any negative effects on their peers. An effective reputation mechanism allows norm-abiding agents to make good peer choices while offering some protection against norm-breakers, by taking advantage of the experience gained through past interactions - either at the individual level (termed *trust*), or at the collective group level (*reputation*).

Our research attempts to pull all of these elements together, by implementing a MAS wherein agents evolve by using social reputation as the sexual selection bias. This preference was not evolved; rather, it was another shortcut that we took. The reasons why reputation is an attractive choice for sexual selection in such a system, at least from the perspective of the system's designer, will become clearer later in this thesis.

1.2 Motivation

The main motivation that led to this research was curiosity. A lot of work has been done on EAs, and a lot on reputation in MAS, yet - to our knowledge - none has been done on the combination of the two. Yet it seemed very likely that there might be a connection - after all, reputation *is* evolved, and it *does* serve mainly as a selection criterion. So the niche was there, and the potential benefits associated with reputation in distributed systems in general could easily apply to distributed evolutionary algorithms.

From a more practical perspective, the author - who comes from an engineering background - has an active interest in intelligent robotics, and in particular, in the application of ANNs as flexible and robust controllers in autonomous systems per-

forming in unpredictable, dynamic environments. Although past results on that front have been encouraging [Chatzinikolaou, 2003], the amount of time required to evolve an ANN controller capable of more sophisticated behaviours on an isolated computer is prohibitive. Finding an efficient way to distribute this computational effort among multiple computers could open the way for the development of much more advanced controllers.

Finally, it can be argued that our findings have also a certain value from a social-theoretical point of view. Although identifying their implications - if any - in that domain is far from the scope of this thesis, it is interesting to see how the concepts of reputation and social selection, intrinsic in social groups of intelligent entities, can potentially shape their evolution.

1.3 Goals

The primary goals that this research set out to achieve are:

- To build an efficient and scalable open software platform capable of distributing the computation effort required for EAs when these are applied to large-scale, complex problems, in networked, cluster and/or multi-core SMP computer systems.

- To investigate the feasibility and performance of a reputation model when used as the selection bias in a distributed EA, in place of the traditional fitness-based approach.

- To investigate the potential benefits of the reputation-based selection bias in "noisy" open distributed environments, wherein agents can potentially be defective, untrustworthy or malicious.

1.4 Contributions

This thesis extends the state-of-the-art in the areas of evolutionary computation, multi-agent systems and reputation in the following ways:

1.4.1 The LiJ LCC Interpreter

The implementation of the *LCC interpreter for Java* (LiJ), although basically a means to an end, can be considered to be an additional contribution of our research. Despite the availability of other LCC interpreters (such as, for instance, the one implemented as part of the OpenKnowledge kernel - see Section 2.3.5 for more details), and even though the current version of LiJ lacks support for networked environments, it offers some substantial advantages that are key for experimental work such as ours.

To begin with, it is very lightweight and has a very small software overhead. Even in its present, unoptimised state (the interpreter was optimised with ease of debugging rather than performance in mind), it was many times faster as an experimentation platform than using the entire OpenKnowledge framework.

It's simple, robust design also makes it extremely reliable; after many CPU years of execution, it always behaved impeccably. This is particularly important in the case of stochastic algorithms such as the ones we were experimenting with, where software bugs can easily go unnoticed - but not without a significant impact to the quality and reliability of the results.

We expect the addition of network support to LiJ to be a straightforward process due to the interpreter's architecture, which would of course increase execution overhead, but without compromising the robustness of the interpreter itself.

1.4.2 A Novel Architecture for a P2P Adaptive GA

The first major contribution of our research was the development of a distributed, P2P adaptive genetic algorithm, designed in the LCC language.

Although the idea of a distributed genetic algorithm is not a new one, and neither is the idea of an adaptive one (see Sections 2.2.5, 2.2.6 and 2.5 for a list of previous work done on these two fronts), to our knowledge there is no previous work that combines both of these elements with an open, peer-to-peer (P2P) architecture. This three-fold combination led to a genetic algorithm that aims to be both parameterless (by virtue of being adaptive) and relatively fast when deployed in a distributed computation environment, due to the improved scalability that a P2P architecture offers.

1.4.3 A Reputation-based GA

A variation of the genetic algorithm described above with which we experimented involved the substitution of direct fitness observation, which is so far the common approach traditionally taken in evolutionary algorithms, with a reputation model. By doing so we were able to confirm our initial hypothesis, that such an approach would yield comparable performance. This investigation led to two positive outcomes:

- First, we show that actual fitness (peer-reported or self-observed) is not the only viable indicator that an evolving entity can consider when deciding on a "mate". Instead, in a social environment with uniform peers, the aggregated "opinions" of all (or even a subset) of these peers on a particular individual can prove to be an equally (if not more) suitable indicator.

- Second, we were able to take advantage of the intrinsic noise resistance that the concept of reputation exhibits, when having to deal with misguiding agents (whether defective or deliberately trying to deceive their peers). This latter point is of particular importance in large-scale, open systems, where there is a lack of an overseeing, regulatory entity, and where the origin (and hence intentions) of peers cannot be known in advance. Although such "defective" agents do take their toll on the overall performance of the system, in our experiments (with up to 50% defective agents) the stability of the system remained intact.

1.5 Roadmap

This thesis is structured in the following way:

- *Chapter 1*, this chapter, introduces the topic, sets the context and presents the motivation and contributions of our research.

- *Chapter 2* presents a broad overview of the three areas in computer science that our work draws upon: Evolutionary computation, distributed systems, and trust and reputation models. In each case we discuss the key concepts that are relevant to our research and provide in-text references to related work. The literature review section at the end of the chapter discusses in more detail a number of recent and/or popular publications related to our work.

- *Chapter 3* offers some insight into the implementation details of our experimentation platform, and discusses the software design of the LiJ interpreter, which is the main software engine of that platform.

- *Chapter 4* discusses a number of issues that relate to the methodology that we followed for conducting our experiments.

- *Chapter 5* comprises the first part of the main corpus of our research, and presents a peer-to-peer, distributed adaptive genetic algorithm implemented in the LCC language and executed using the LiJ interpreter, along with the behavioural and performance results obtained from a number of experiments.

- *Chapter 6*, the second main part, discusses the extension of the algorithm by replacing direct fitness observation with various trust and reputation models. Again, results from multiple experiments are given that illustrate how the extended algorithm behaves and performs.

- *Chapter 7* concludes this thesis by restating the contributions of our research, and how these relate to our initial goals. In addition, it includes number of ideas and suggestions for extending this research further in the future.

Three appendices at the end of the thesis provide additional information that allows future research to replicate our results. These are:

- *Appendix A* is a concise user's guide for writing and executing LCC protocols using the LiJ interpreter, along with a few examples that illustrate how LCC can be used to write MAS interactions.

- *Appendix B* lists the source of the LCC protocols that implement the evolutionary MAS algorithms used in our experiments.

- *Appendix C* lists the Java source code that provides the context experimentation software, including the implementations of the constraint methods used in our LCC protocols.

The source code and binaries for the LiJ interpreter itself, along with a number of example LCC interaction models, can be downloaded from the project's homepage on SourceForge, at `http://sourceforge.net/projects/lij/`.

For the implementation of the evolutionary functionality, the home-brewed *Java library for Evolutionary Algorithms* (JEvA) was used. The library binaries as well as its source code can be found at `http://sourceforge.net/projects/jeva/`.

Chapter 2

Background

2.1 Overview

In this chapter, we take a look at the three fundamental areas that our research touches on: Evolutionary algorithms, distributed computing, and trust/reputation. For each of these topics, in addition to an overview of the fundamental concepts, we provide references to related work in the literature, and identify and justify the particular approaches that we chose to follow in our research.

In the topic of evolutionary algorithms, we present the general categories in which these are usually divided, discuss their applications and limitations, and explore common approaches for dealing with the latter.

We then proceed to present an overview of the whys and hows of distributed computation, including a look at some of the main types and their similarities and differences.

Further, we continue with some insight into trust and reputation models typically used in distributed systems, including their definitions and characteristics.

Finally, we conclude this chapter with an itemised literature review, providing a summary of the related literature.

2.2 Evolutionary Computation and Genetic Algorithms

2.2.1 Evolutionary Algorithms

Broadly speaking, evolutionary computation is an area of artificial intelligence that aims to optimise combinatorial problems using the paradigm of natural evolution, or -

in other words - the Darwinian principle of the survival of the fittest.

The beginnings of evolutionary computation date back to the 1960s, with the pioneering work of Nils Aall Barricelli on artificial life [Barricelli, 1962] generally considered to have given birth to the field. Since then, a lot of progress has been made in this area, and now the term applies to various sub-categories of evolutionary algorithms (EA). An attempt to present a comprehensive review of the existing literature would result in a book of multiple volumes. As this is outside the scope of this thesis, we will instead give a brief overview of the main types of EA. These are:

- Genetic Algorithms (GA)
  A GA is an iterative search method inspired by natural evolution [Holland, 1975]. A GA consists of a population of candidate solutions to a problem, which is defined by a fitness function, originally consisting of random values. Each subsequent population in each GA iteration improves on the previous one, by using genetic operations such as selection, crossover and mutation, on the individuals in the population. We discuss GAs in more detail in the following section.

- Evolution Strategies (ES)
  ESs, invented by Ingo Rechenberg [Rechenberg, 1971], are very similar to GAs, to the extent that a number of publications exist that attempt to explain their differences (e.g., [Hoffmeister and Bck, 1991, Okabe et al., 2005]). Even though they were developed independently from GAs, their similarities far outnumber their differences. Their main difference lies in performance (ESs are generally faster for "good-enough" solutions, while GAs are generally better at finding the global maximum), and parameter encoding (ESs use real number parameters, while GAs use bitstring representations). For all practical purposes, however, the two types have converged to mean the same thing.

- Evolutionary Programming (EP)
  In EP, introduced by [Fogel, 1962], the objective is to optimise a fixed program by allowing its numerical parameters to evolve. A similar, more extended variation of this is Genetic Programming (GP), where the individuals in the evolving population represent entire programs rather than simply numerical parameters. In both cases, the fitness of a candidate solution is determined by the ability of the resulting program to perform a given task.

- Swarm Intelligence (SI)

SI algorithms, although also inspired by natural systems and organisms, differ significantly from the other three types in that they do not rely on genetic operations (selection, crossover and mutation) to evolve individuals in a population. Instead, they are based on populations of agents of relatively simple isolated behaviours, which are able to interact with each other and with their environment. Their evolutionary properties stem from the resulting emergent behaviour of the system as a whole. A very popular SI variation is Ant Colony Optimisation (ACO) [Maniezzo and Carbonaro, 1999, Dorigo and Stützle, 2004], which is particularly useful in path and route optimisation problems. ACOs are based on simple "ant" agents that, like real ants marking their path using pheromones, record their solutions in the search space using similar simulated techniques.

In our research, we have borrowed ideas and techniques from all of these types of EA. For instance, we use both bitstring parameter representations (GA) and real value representations (ES). The interactions between our agents conform to fixed social norms (SI), and in Section 7.2.4 we hint towards a future generalisation of our system that could evolve arbitrary parameterisable programs (EP/GP). For practical purposes, we will refer to our agent system as a Multi-agent System (MAS) rather than a SI system, and to its constituent subsystems as GAs, since this is the more general of the different EA types.

2.2.2 Genetic Algorithms

Since their inception by John Holland in the early 70's [Holland, 1975] and their popularisation over the last few decades by works such as [Goldberg, 1989], GAs have been used extensively to solve computationally hard problems, such as combinatorial optimisations involving multiple variables and complex search landscapes.

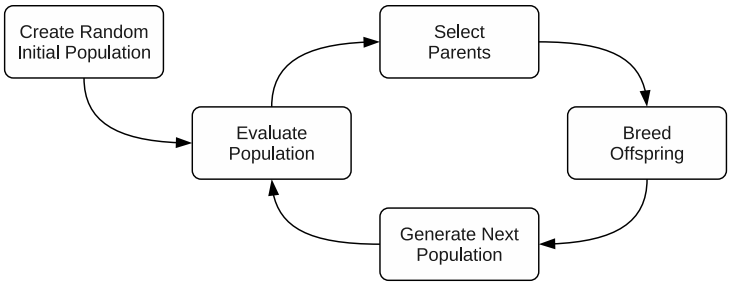


Figure 2.1: The operation cycle of a typical genetic algorithm.

In its simplest form, a GA is a stochastic search heuristic that operates on a population of potential solutions to a problem, applying the Darwinian principle of survival of the fittest in order to generate increasingly better solutions. Each generation of candidate solutions is succeeded by a better one, through the process of selecting individual solutions from the current generation according to their relative fitness, and applying the genetic operations of crossover and mutation on them to produce offspring.

The result of this process is that, over a period of time, latter generations consist of solution approximations that perform better than their predecessors, just as is natural evolution.

2.2.3 Applications

GAs have proved to be flexible and powerful tools, and have been successfully applied to solve problems in domains too numerous and diverse to exhaustively list here. Some particularly popular applications, however, are:

- Economics and optimisation of financial models in the stock market [Marney et al., 2001, Wang, 2006].
- Scheduling [Gonalves et al., 2002, Page and Naughton, 2005].
- Classification [Espejo et al., 2010, Robu and Holban, 2011].
- Computer network optimisation and security [Venketesh and Venkatesan, 2009, Owais et al., 2008].
- Software testing [Berndt and Watkins, 2005, Rathore et al., 2011].
- Learning and optimisation for Artificial Neural Networks (which was also our own initial motivation, as explained in Section 1.2) [Abbass, 2002, Herzog et al., 2009].
- Automated Computer-aided Design (CAD) in engineering and manufacturing [Li et al., 2004, Kureichik et al., 2009].
- Analogue and digital electronic circuit design [Das and Vemuri, 2007, Ashraf et al., 2012].
- Power electronics [P.N.Hrisheekesha and Sharma, 2010].

- Routing and layout optimisation [Ramakrishna, 2002, Hong et al., 2005].
- Materials engineering [Paszkowicz, 2009].
- Chemistry and bioinformatics [Gondro and Kinghorn, 2007, Wong et al., 2010].
- Medicine [Xin et al., 2012, Ghosh and Mitchell, 2006, Stylios and Georgopoulos, 2008].
- Geophysics [Ramillien, 2001].
- Security systems and identification [Tan and Bhanu, 2006, Ammar and Tao, 2000].
- Image processing [Zhao et al., 2008, Assudani and Malik, 2012].

A very extensive literature collection, updated regularly and including many thousands of related publications, can be found in `http://delta.cs.cinvestav.mx/~ccoello/EMOO/EMOObib.html`.

2.2.4 Limitations

Despite the widespread success of GAs, there's still a number of issues that make their deployment by the uninitiated a non-trivial task. Two themes that keep recurring in the literature are:

- Parameter control
  This involves determining the optimal set of parameters for a GA.
- Parallelisation
  This involves distributing the computational load of a GA between multiple computational units.

It is on these two themes that this research concentrates. In the following two sections we will describe in more detail these two problems, along with the most common approaches for dealing with them.

2.2.5 Adaptation in GAs

In every application of a GA, the designer is faced with a significant problem: tuning a GA involves configuring a variety of parameters, including things such as population sizes, the operators used for selection and mutation, type and size of elitism etc. As a general case, before a GA can be deployed successfully in any problem domain, a significant amount of time and/or expertise has to be devoted to tuning it.

As a result, numerous methods on parameter optimisation have appeared over the years [Eiben et al., 2000]. These generally fall in one of two categories:

- Parameter Tuning, in which the set of GA parameters are determined a priori, and then applied to the GA before it is executed.
- Parameter Control, in which the parameters change (adapt) while the GA is running.

It was discovered early on [Hesser and Männer, 1991, Tuson, 1995] that simple a priori parameter tuning is generally insufficient to produce adequate results, as different stages in the evolutionary process are likely to require different parameter values. Therefore, in our research we concentrate on dynamic parameter adaptation, along the lines of work presented in [Back, 1992, Eiben et al., 2000, Meyer-Nieberg and Beyer, 2006].

2.2.6 Parallelising GAs

Even after a set of optimal parameters has been established, traditional (canonical) GAs suffer from further difficulties as problems increase in scale and complexity. [Nowostawski and Poli, 1999] has identified the following:

- Problems with big populations and/or many dimensions may require more memory than is available in a single, conventional machine.
- The computational (CPU) power required by the GA, particularly for the evaluation of complex fitness functions, may be too high.
- As the number of dimensions in a problem increases and its fitness landscape becomes more complex, the likelihood of the GA converging prematurely to a local optimum instead of a global one increases.

To some extent, these limitations can be alleviated by converting GAs from serial processes into parallel ones. This involves distributing the computational effort of the optimisation between multiple CPUs, such as those in a computer cluster.

The approach of parallelisation of genetic algorithms becomes even more appropriate in the light of recent developments in the field of multi-processor computer systems [Munawar et al., 2008], as well as the emergence of distributed computing, and particularly the new trend towards cloud computing [Foster et al., 2008].

[Lim et al., 2007] identify three broad categories of parallel genetic algorithm (PGA):

- Master-slave PGA

  This scheme is similar to a standard, or canonical, genetic algorithm, in that there is a single population. The parallelisation of the process lies in the evaluation of the individuals, which is allocated by the master node to a number of slave processing elements. The main advantage of master-slave PGAs is ease of implementation. However, as a centralized scheme, it suffers from scalability issues as the number of processing nodes increases. In addition, the existence of a single-point-of-failure (the master node) detracts from such a system's robustness.

- Fine-grained or Cellular PGA

  Here we have again a single population, spatially distributed among a number of computational nodes. Each such node represents a single individual (or a small number of them), and the genetic operations of selection and crossover is restricted to small, usually adjacent groups. The main advantage of this scheme is that it is particularly suitable for execution on massively parallel processing systems, such as a computer system with multiple processing elements.

- Multi-population or Multi-deme or Island PGA

  In an island PGA there are multiple populations, each residing on a separate processing node. These populations remain relatively isolated, with "migrations" taking place occasionally. The advantages of this model is that it allows for more sophisticated techniques to be developed.

The list of parallelisation schemes given above is not exhaustive. Among others, [Cantu-Paz, 1998, Nowostawski and Poli, 1999, Alba and Troya, 1999] each provide an excellent coverage of the work done on this theme.

The work presented in this thesis was originally influenced by [Arenas et al., 2002], which follows the island paradigm [Tanese, 1989, Belding, 1995]. In our system, however, we combine this with the cellular PGA scheme, by means of a cascaded meta-GA.

2.3 Distributed Computing and Multi-agent Systems

2.3.1 Why Distribute Computation

Distributed computing in general refers to the practice of allocating a (normally hard, computationally expensive) task across multiple independent computational nodes, which are able to communicate between them in order to coordinate towards that common goal.

Distributed computing systems reach further than parallelising GAs. Such systems are commonly used for diverse applications, ranging from lengthy scientific/research simulations (as in our case), to telecommunications, data mining and computer graphics rendering farms.

The last few years have seen a trend towards cloud computing, which is essentially the move of data and computation from individual computers to multiple shared ones, scattered across a network. Similar models of distributed computation, albeit with less catchy names, include grid computing and cluster computing. Some notable current examples of commercial cloud services geared towards everyday users include Amazon's AWS, Google's Apps and Salesforce's cloud CRM.

In addition, in recent years CPU manufacturers have hit a ceiling in their attempts to conform to Moore's law, as they have reached the point where the size of individual transistors and junctions in integrated circuits (IC), key to performance, approach molecular sizes. For instance, Intel's top-of-the-line CPUs at the moment of writing this employ 22 nm technology (dubbed "Ivy Bridge"), leaving little space for progress towards further minimization. Instead, CPU trends move towards extending Symmetric Multi-processing (SMP) designs, where each CPU IC contains multiple CPU cores (with shared memory but independent caches). Many desktop computers sold today, and even a few smartphones, use dual-core CPUs, with top-of-the-line computers sporting up to eight. Prototype CPUs not yet in commercial production (such as the one based on [Chalamalasetti et al., 2009]) feature as much as 1000 cores on a single chip. Although not technically a distributed system, applications can use similar architectures in order to utilise SMP ICs to speed up tasks.

Apart from distributing computation load, there is another advantage to distributed systems: That of robustness and redundancy [Georgiou et al., 2005]. Assuming an appropriate design methodology (e.g., Peer-to-peer - see the following sections), such systems benefit from the lack of a single-point-of-failure. The main implication of this is that, should one or more nodes in the system break down, the system as a whole stands a better chance of being able to continue to function using the remaining, operating nodes. It is hard to imagine such a scenario with a SMP system, where usually a malfunctioning core will bring about the failure of the entire IC; however, the lack of a single-point-of-failure becomes particularly important in networked (grid/cluster/-cloud) systems, where communication errors due to misbehaving networking are more likely to occur, and individual node hardware remains more-or-less isolated.

Of course, apart from advantages, distributed systems have their share of detractors. The most important of these is the increased factor of complexity - after all, it is a much more involved process to design software for a distributed system than it is for a single machine. Several approaches have been proposed to facilitate this process. In the following section, we are going to have a look at some of the most relevant ones.

2.3.2 Types of Distributed Systems

Three commonly encountered types of distributed systems are the following:

- Grid Systems

  A grid computing system comprises a collection of usually diverse and geographically distributed computers, which can collectively be treated as a single, virtual supercomputer. It is generally aimed towards high-performance applications involving large-scale sharing of data and computation resources [Foster et al., 2001]. Grid systems are relatively fixed in scope, operating in highly controlled conditions and with each node generally having clearly defined roles and access rights. Two of the most powerful grid systems at the moment are *BOINC* [Anderson, 2004] and *Folding@home* [Larson et al., 2009].

- Peer-to-peer (P2P) Systems

  The distinction between a grid system and a P2P system can be fuzzy at times, as the two types tend to converge in their basic aspect of resource sharing organisation [Foster and Iamnitchi, 2003]. Indeed, grid systems often use P2P technology for things such as service discovery [Caron et al., 2009]. Treated individually,

however, their main difference is that P2P systems are generally more loosely-coupled systems, with symmetrical participating nodes and no central arbitrator entity. In addition, they do not adhere to fixed topologies, hence nodes are able to self-organise into sub-topologies according to the prevailing conditions at any one time [Androutsellis-Theotokis and Spinellis, 2004]. Another desirable characteristic of P2P systems is improved scalability, which stems from reduced bottlenecks [Han, 2004, Bondi, 2000]. Finally, P2P systems, unlike grid systems, are generally designed to address failure rather than infrastructure [Foster and Iamnitchi, 2003]. Notable P2P systems include *Gnutella* [Ripeanu, 2001] and *Bittorrent* [Chow et al., 2009].

- Multi-agent Systems (MAS)
  Agent-based computing concerns the development of complex applications by means of a number of autonomous software agents, capable of interacting with each other in order to solve a common task [Luck et al., 2004]. Typical MAS systems involve agents with pro-active, intelligent behaviours, as in the Belief-Desire-Intention (BDI) architecture. They are able to communicate and coordinate using a variety of languages and protocols, such as the ones standardised by the Foundation for Intelligent Physical Agents (FIPA). Agents are often mobile, i.e. they are able to relocate themselves on different computation nodes of the network. This results in potential savings in network resources, increased performance and dynamic reconfigurability [Fortino and Russo, 2008]. As decentralised systems, the benefits in scalability pertaining to P2P systems also apply to MAS. Unlike typical P2P systems, however, agents in a MAS do not necessarily behave uniformly, and can instead assume varying roles. The resulting emergent behaviour of the system as a whole is what lends MAS their - collective - intelligence [Deguet et al., 2007, Rzevski and Skobelev, 2007]. Again, the distinction between MAS and P2P systems is not set in stone, as various MAS are based on P2P architectures for interoperation (e.g., [Panti et al., 2002, Huang, 2010, Robertson et al., 2006]. According to others (e.g., [Brzykcy, 2009]), MAS can conversely be considered a superset of P2P systems.

Our particular case falls somewhere between these last two types: Our system can broadly be described as a multi-agent system, based on a peer-to-peer architecture. The reason is that, intrinsically, this scheme most readily lends itself to the cellular/island hybrid GA parallelisation approach that we follow, as mentioned in Section 2.2.6.

2.3.3 Open Multi-agent Systems

[Jennings, 2001] identifies a number of properties that individual agents in a MAS must exhibit. These are:

- An agent is an identifiable problem-solving entity, having clearly defined boundaries and interfaces.
- Agents have partial control and observability over their environment (including their peers).
- Each individual agent is designed with a particular role/objective in mind.
- Agents are autonomous, in that they have control over their internal state and behaviour.
- An agent must exhibit a certain level of flexibility, in pursuing its objectives. It can be both reactive to changes in its environment, as well as proactive in taking initiatives.

When we refer to an open distributed system, we generally mean one in which no concrete assumptions can be made about the participating entities; in particular, about their topology, platform, and evolution of behaviour [Cruz and Ducasse, 1999].

In the context of an open MAS, this translates to the following additional characteristics [Huynh et al., 2006, Barber and Kim, 2003]:

- The environment becomes dynamic, in the sense that new agents can enter an interaction at any time, while existing ones may be removed.
- Following this, the number of agents in an open MAS is unbounded.
- There is a lack of security; in particular, agents in the system may be defective, counter-productive or even malicious.
- Agents are unable to have complete knowledge of their environment, as this would be impractical in large-scale applications.
- There is no central arbitrator entity, which may mean that each agent is self-interested and follows its own agenda.

From the above, it can be deduced that the cost of openness in a MAS is not negligible. At the very least, it necessitates a common communication standard between agents (discussed in the following two sections), as well as a mechanism for trust (discussed in Section 2.4.

2.3.4 Electronic Institutions

The concept of an electronic institution [Esteva et al., 2000] is now standard in MAS research. The basic idea is, using a standard language, to provide a specification of the interaction desired within a community of agents. Since the interaction specification is standard it then becomes possible for agents to reason about interactions and, conversely, for the interaction specifications to be used to constrain participation in collective activities.

Many different styles of specification are proposed for electronic institutions, these legitimately differing depending on the style of deployment. A good survey on these can be found in [Horling and Lesser, 2004]. In our particular case, we are interested in potentially deploying highly parallelized GAs in large scale and open distributed environments. Therefore, we chose as a specification language the Lightweight Coordination Calculus (LCC) [Robertson, 2004a, Robertson, 2004b].

2.3.5 OpenKnowledge and The Lightweight Coordination Calculus

LCC is a process calculus in which one can specify the different roles in an interaction, with synchronization between roles through message passing. It is an executable specification language so definitions in LCC can, with appropriate interpreters, be used to enact as well as to specify interactions between agents. A variety of interpreters have been written for enactment in different computational architectures, the most relevant of which is the P2P system developed as part of the OpenKnowledge framework [Robertson et al., 2006].

The kernel of the OpenKnowledge system, deployed on each agent, contains an interpreter for LCC that communicates with the agent to inform it of the constraints it must satisfy to participate in the roles in which it has chosen to participate in the interactions within which it is engaged. The kernel can also relay data on performance of the agent in interactions - this being used to provide various methods for analysis of reputation.

In the following chapters of this thesis we shall describe experiments with one form

of reputation and its use in supporting agents that are running evolutionary algorithms. The OpenKnowledge system operated on top of a third-party peer-to-peer infrastructure. Our experiments, although not run on that infrastructure, are based on the same assumptions of independence, parallelism and non-interference between agents.

2.4 Trust and Reputation

2.4.1 The Need for Trust in Open Systems

One definition of *trust* generally accepted in the literature is the one given by [Abdul-Rahman and Hailes, 2000]:

> *Trust is a measurable level of the subjective probability with which an agent a assesses that another agent b will perform a particular action in a favourable way to a, both before a can monitor such action (or independently of its capacity ever to be able to monitor it) and in a context in which it affects its own action.*

As explained in Section 2.3.3, the need for a trust mechanism arises from combining a (decentralised and unregulated) P2P system with agents that are not guaranteed to be uniform (MAS) in an open system, where participants may be of unknown origin. Such participants may be defective, unreliable or plain malicious, which - left unchecked - would prove detrimental to the performance and stability of the system as a whole.

It can be argued that, in our particular case, where experiments were conducted in predictable, tightly controlled environments, there is no real need for trust. This argument would be completely true - after all, all our agents were clones (hence uniform), there was no network to begin with (hence the system was open only in design), and we were the sole providers of these agents (hence of known origin).

This, however, would be missing the point of this research entirely. Our objective, as mentioned in Section 1.3, was the implementation of a platform architecture, as opposed to an application. The characteristics of an open, P2P MAS were specifications rather than solutions. And, most importantly, by adhering to such an architecture, we were able to test our initial hypothesis, namely, that reputation in such a system can guide evolution.

2.4.2 Approaches for Trust and Reputation

[Huynh et al., 2006] identifies two broad categories of approaches to trust. These are:

- Cognitive trust

  The cognitive view on trust [Falcone and Castelfranchi, 2001] takes a high-level approach towards estimating the trust that an agent *a* has towards another agent *b*. Following the cognitive paradigm of BDI architectures, trust is evaluated as a function of the beliefs that *a* holds towards *b*, such as beliefs about *b*'s competence, willingness, persistence and motivation. Although this approach has the benefit of a more natural modelling expression, it is often impractical in its implementation. This is due to the fact that it is rarely possible for *a* to have a reliable (if any at all) model of *b*'s behaviour and intentions.

- Probabilistic trust

  The probabilistic approach to trust [Yu and Singh, 2002] differs significantly from the cognitive one in that it relies on the actual experiences of the agents in the system, obtained through observation rather than speculation. It is still uncertain whether a trusted agent will perform as required, however it assumed that a more-or-less consistently positive performance of an agent in the past is a fairly reliable indicator about its performance in the future. In addition, and unlike the cognitive approach, it is fairly straightforward to design agents that can observe and record such past experiences with their peers. This makes the probabilistic approach to trust much more practical in the context of open MASs, and is hence the one we follow in our research.

The definition of probabilistic trust given above refers to a single agent's self-observed experiences between itself and its peers (termed *direct trust* by [Huynh et al., 2006]). Although in theory (as well as in practice, as we found out in our preliminary investigations) this approach can by itself provide some protection against ill-behaving peers, it is limited in scope by the fact that an agent's knowledge about a peer's past performance is restricted to experiences obtained by that agent alone. In the case of peers with which the agent has no previous experience, this knowledge will be none at all, which would leave the agent at a higher risk of choosing to interact with an unreliable peer.

An alternative take to this is to have the agent initiating the interaction exploit the collective experiences of other peers in the system, either instead of or in addition to

direct trust as defined above. This is the general notion behind the term *reputation*, which is defined by [Abdul-Rahman and Hailes, 2000] as:

> *A reputation is an expectation about an agents behaviour based on information about or observations of its past behaviour. Reputational information need not solely be the opinion of others. We also include reputational information completely based on an individual agents own personal experiences. This allows us to generalise reputational information to combine personal opinions and opinions of others for the same reputation subject.*

In our research, we followed this approach by implementing a combined reputation model, wherein direct (self-based) trust is used by agents for selecting peers for suggestions, and the collected external (peer-based) reputational information is used for selecting peers for interactions.

This particular combination has a two-fold advantage: First, it deals to some degree with the intrinsic *subjectiveness* of suggestions, as it draws them from peers that have been proven - in an agent's own past experience - to be compatible with that particular agent. Second, it provides some resistance against *Sybil* attacks [Douceur, 2002], assuming that a direct interaction with a malicious agent would probably not be beneficial to the agent in the first place, and hence would prevent the malefactor from being chosen as a (misguiding) suggester in future interactions.

2.5 Related Work

In this section we review a number of papers that are related to this thesis, either by having served as foundations for our work, or by presenting alternative approaches and hence providing context for our research.

This literature review has been split into two sections, one for each of the broader AI fields that our work borrows from: Distributed Adaptive EAs, and Trust and Reputation Mechanisms.

2.5.1 Distributed and/or Adaptive Evolutionary Algorithms

[Back, 1992]

In one of the earliest works on GA parameter adaptation, [Back, 1992] propose a solution based on Evolutionary Strategies (ES) in which the parameter to be adapted (in

this case, the mutation rate) is not exogenous to the solution genome, but instead constitutes a part of it - i.e. the mutation rate is encoded and becomes part of the genome being evolved.

[Back, 1992] found that it is possible to achieve automatic parameter control using this method, by using n mutation rates per individual for multimodal fitness functions, or a single mutation rate per individual for unimodal ones.

The results presented in this paper were an encouraging starting point, although this approach suffers from a number of issues, particularly premature convergence of the mutation rate values past the threshold of effectiveness [Glickman and Sycara, 2000].

The self-adaptation technique presented in [Back, 1992] showcases the main alternative to our meta-GA approach, in one of its earliest implementations.

[Murata et al., 2007]

In [Murata et al., 2007] (which builds on [Takashima et al., 2003]), the authors propose a two-layer meta-GA architecture based on the island PGA paradigm, called "Self-Adaptive Island GA" (SAIGA). In their work, a lot of emphasis has been placed in adapting multiple GA parameters at the same time.

Multiple populations of candidate solutions are distributed among a fixed number of different "island" agents, placed in a ring topology. At the lower level of the meta-GA, each agent executes a standard GA, which uses a vector of parameters. These parameters are the population size, mutation rate, crossover rate, and tournament size.

After a predetermined number of GA evaluations (termed an "era"), the relative increase in fitness in each island is compared, and these gains are used as the fitness values of the parameter vectors themselves. In this way, the parameter vectors for the lower-level GA can be optimised.

[Murata et al., 2007] did tests using the Travelling Salesman Problem (TSP), a deceptive problem, as well as a 10-variable version of the Rastrigin function. Their results were particularly promising, showing improved performance compared to a "default" GA which uses De Jong's rational parameters. Performance was also close to a hand-tuned (non-adaptive) canonical GA.

[Murata et al., 2007] (as well as earlier work by the same authors) provides one of the foundations for our own system, especially regarding the P2P adaptive GA we present in Chapter 5, which is also based on the meta-GA paradigm.

[Dreżewski et al., 2009]

[Dreżewski et al., 2009] implemented a distributed GA based on a multi-agent paradigm, where each agent, essentially and "island", works on the same problem in parallel with its peers. Agents are able to communicate and interact with their peers, in order to perform crossover between them.

This system is based around the concept of limited resources and resource allocation between agents. Each agent is allocated a share from a finite amount of resources, which it uses in order to perform operations such as crossover and migration between a peer. Agents performing poorly are allocated less resources and vice versa, which results in resource competition between agents, and hence an increase in the system's overall performance.

The results reported show an improvement in solution quality (for the TSP problem), although at the cost of longer execution times.

The main difference between our P2P adaptive GA and the system presented in [Dreżewski et al., 2009] is that the latter does not concentrate on any form of adaptation or parameter control. On the contrary, the architecture the authors propose relies on a large number of configuration parameters, including the standard GA ones (population size, mutation and crossover rates), values for migration frequency and maximum agent lifetime, resource limits, as well as costs for migration, mutation and crossover.

[Lim et al., 2007]

[Lim et al., 2007] propose a distributed GA architecture based on grid computing, which they term "Hierarchical Parallel Genetic Algorithm" (GE-HPGA). Their system is based around two fundamental features: An Application Programming Interface (API) for abstracting the grid platform's complexity, and a meta-scheduler for resource discovery and selection.

The work in this paper focuses more on distribution of computation, hence there is no parameter adaptation involved. Instead, most nodes in the computing grid undertake solely the evaluation of sub-populations, whereas the task of performing evolutionary operators such as mutation, crossover and selection is left to "sub-population evolution" nodes. The whole process, including the migration phase, is handled by an additional, central node (the "master" node).

The performance of this system was measured by using a benchmark function (a 10-variable version of the Rastrigin function) as well as a realistic aerodynamic airfoil

shape optimization problem. The results presented show a significant speed-up in performance, although - due to the distributed nature of the system - this speed-up depends on fitness function cost, computing cluster size and communication overheads.

This paper shows a distributed GA architecture fundamentally different from our own in that, apart from the lack of adaptation, it also follows a centralised model.

[Krink and Ursem, 2000]

The work presented in [Krink and Ursem, 2000], although somewhat dated, is included here as an interesting variation on the Terrain-Based Genetic Algorithm (TBGA) [Gordon et al., 1999], itself a self-adaptive Cellular Genetic Algorithm (CGA).

In TBGA, each individual in the evolving population is positioned on a two-dimensional parameter lattice, where its location dictates the mutation and crossover rates used for that particular individual. The location of each individual remains fixed, and only neighbouring individuals can mate with each other. This approach ensures that, at any time, there are at least a few individual with optimal parameters for that particular problem in that particular stage in evolution. On the flip side, the fixed position of the individuals on the parameter grid means that only a small proportion of those individuals can benefit from the optimal parameters.

[Krink and Ursem, 2000] tackle this issue by incorporating their agent-based Patchwork model in the architecture. They treat individuals as autonomous agents that are able to move about the parameter space, and that are allowed to occupy the same position at the same time. This characteristic of mobility in the agents makes the system similar to "island" based parallel GAs.

These agents act on "desires", e.g. to keep close to high-fitness peers, move away from empty spaces, avoid over-crowding, mate etc. A motivation network model controls their behaviour according to their location and state, the state of their peers, and the environment.

[Krink and Ursem, 2000] tested their architecture against the original TBGA as well as their original implementation of the Patchwork model, using five different benchmark functions. The results they present show an improvement in performance over both the original constituent models for at least some of these benchmark functions, and no drop in performance for any of them.

Even though this architecture manages to adapt crossover and mutation rates, it does introduce four additional parameters relating to agent behaviour - excluding the motivation network model itself. However, [Krink and Ursem, 2000] found that the

optimal choice of these parameters appeared to be independent of the test problems.

Apart from the multi-agent paradigm, our work does not share much with [Krink and Ursem, 2000] - however, the latter is included here as a demonstration of the multitude of different ways in which parameter adaptation can be achieved in GAs.

[Clune et al., 2005]

In this investigation, [Clune et al., 2005] attempt to verify the three basic claims of adaptive meta-GAs, in particular whether they are able to:

- Adapt their genetic operator parameters accordingly for different problems, and thus perform well on average across diverse problems.
- Perform this adaptation throughout the lifetime of the GA, thus maintaining good parameter values across different stages in evolution.
- Converge to optimal parameter values for a given problem.

[Clune et al., 2005] performed experiments using a standard meta-GA and two optimisation problems: The "counting ones" problem, and a 4-bit "deceptive trap". They contrasted the performance of the meta-GA against specialist fixed GAs that were hand-tuned for each problem.

The results they present show that the first claim is generally true, as the meta-GA was indeed able to perform almost as well as the specialist GA in each problem, without requiring a-priori hand-tuning. The results for validating the other two claims were not as encouraging, though. [Clune et al., 2005] found that the meta-GA is indeed able to switch strategies mid-evolution as expected (for the trap problem), but this switch did not in fact improve the performance of the meta-GA - possibly because it got stuck in local optima in the parameter space. Also, they found that the final parameters discovered by the meta-GA were again not the best, possibly for the same reason.

From these findings, [Clune et al., 2005] conclude that a meta-GA approach to adaptation can indeed be effective, but should only be used when a general-purpose, parameterless optimiser is required, and the human investment required to set up individual runs is more precious than performance.

As was the case with [Murata et al., 2007], the meta-GA approach in [Clune et al., 2005] is similar (more in principle than in implementation) to what we followed for our own adaptive GA. Although [Clune et al., 2005] do not contrast the performance

of their meta-GA with any alternative adaptive EAs, they do a good job on identifying the limitations of this approach.

[Eiben et al., 2006]

In [Eiben et al., 2006], the authors propose a method for boosting GA performance using self-adaptation, which is however limited to selection pressure - unlike most other works in the field, where more "fundamental" parameters such as population size and mutation and crossover rates are adapted.

They attempt this using two approaches: "Pure" self-adaptation, and "hybrid" self-adaptation (HSA). Both approaches require a (centralised) mechanism that gathers and aggregates the "votes" from each individual regarding tournament selection size.

In the "pure" version, the vote of each individual is included as a single bit in that individual's chromosome. The final (global) tournament size is defined as the sum of all individual votes.

In the extended HSA version, instead of randomly mutating the "vote" allele of each individual, a heuristic rule is applied that regulates the mutation more intelligently. In particular, those individuals that are better than their parents aim to increase selection pressure (since this will give it an overall advantage over less fit individuals), and those that are worse tend to lower it.

The results given in the paper show that varying selection pressure on-the-fly significantly improves the performance of the GA, with the hybrid HSA approach yielding even better results in smoother solution landscapes.

The drawback of this system, as is common in adaptive GAs, is that even though it does away with one parameter (tournament size), it introduces another one (γ, the learning rate used for controlling the adaptation speed). However, [Eiben et al., 2006] claim that, typically as well as specifically for their case, such meta-parameters are less accuracy-sensitive than the actual, "technical" GA parameters, in addition to remaining more-or-less identical for different optimisation problems run on the same system.

The adaptation mechanism presented in this paper is closer to the self-adaptation approach (as in, e.g., [Back, 1992]) than the meta-GA approach that we followed in our work. Despite this fundamental difference, it demonstrates that the inclusion of even simple heuristic rules can result in improved performance. In our case, such heuristic rules were deliberately and emphatically avoided, since our research concentrated on investigating a principle rather than pushing for performance.

[Yuan, 2005]

In this paper, [Yuan, 2005] propose a hybrid self-adaptive GA approach that combines a standard meta-GA with another technique for parameter control, called "Racing" [Maron and Moore, 1997].

[Yuan, 2005] list a number of limitations that meta-GAs suffer from when it comes to self-adaptation:

- Some parameters are not directly searchable, since there exists no obvious distance metric along their dimension (e.g. selection operator type, crossover type etc.)
- Some parameters are only applicable in specific combinations (e.g. tournament size applies to tournament selection, but not truncation selection).
- Meta-GAs are typically time-consuming, since they require a lot of computation during the tuning phase (here [Yuan, 2005] refer to parameter tuning as opposed to parameter control - see 2.2.5).

The architecture proposed in [Yuan, 2005] attempts to address these limitation by combining their meta-GA with the "Racing" statistical method.

In its original form, "Racing" is a search method for identifying the best learning method among a set of candidates. It works by performing a number of tests of the candidates in parallel, quickly identifying weak individuals, and concentrating computation effort on the stronger ones. Its main advantage in the context of a meta-GA is that it is independent of the internal structure of the learning algorithms, which implies that (a) the latter can be non-uniform, and (b) a distance metric is not required for the parameters.

In the system presented in this paper, the meta-GA is responsible for optimising the tunable parameters, whereas Racing was employed to identify the best algorithm from a set of candidates that differed in terms of the non-tunable ones.

[Yuan, 2005] performed tests using the "One-Max" and the "Hierarchical-If-and-only-If" benchmark problems, and found that the performance of the meta-GA was significantly improved with the addition of Racing.

In our case, we followed a pure meta-GA approach and did not include Racing, in order to keep things simple. It is interesting however to note that, by using techniques such as Racing, the performance of meta-GAs can be improved even further.

[Seredynski et al., 2003]

In [Seredynski et al., 2003], the authors propose a multi-agent parallel coevolutionary algorithm, called "Loosely Coupled Genetic Algorithm" (LCGA), and compare its performance against another well-known coevolutionary algorithm, "Cooperative Coevolutionary Genetic Algorithm (CCGA) [Potter and Jong, 1994], as well as a canonical, sequential GA.

The principle of co-evolution in the context of function optimisation is based on the notion that it is preferable/more realistic to co-evolve a number of different *species*, each representing a part of the global solution, rather than have a single population consisting of the same species.

CCGA works by treating each variable as a separate species, each evolving in its own subpopulation. This approach requires that, in order to evaluate the fitness of each individual belonging to a species, it must be combined with individuals from all other species. To achieve this, a two-phase synchronisation mechanism is required. In the first phase, each subpopulation is evaluated in turn while the other subpopulations remain frozen (in a round-robin fashion). As a result, CCGA is a partially parallel, centralised GA.

The LCGA algorithm is motivated by non-cooperative game theory models. In LCGA, each variable is again considered an individual co-evolving species, represented by an agent. These subpopulations, using game-theoretic mechanism of competition, act to maximize their local goals described by local functions. In LCGA, evaluation can take place in a distributed fashion, by deriving these locally defined fitness functions. This is achieved by first analysing the problem to be solved in terms of its possible decomposition and relations between subcomponents, expressed by a problem defined communication graph called a graph of interaction.

[Seredynski et al., 2003] performed tests using a number of benchmark functions, including some from De Jong's suite and the Rastrigin function among others. The results they report show that LCGA performs well for those cases where the global goal of the system is the sum of local goals - i.e. unimodal problems such as De Jong's Sphere function, as well as multimodal problems expressed as a sum of local functions (such as the Rastrigin function).

As in our case, the system presented in [Seredynski et al., 2003] follows the evolutionary multi-agent paradigm, but builds on it with co-evolution and game theory. Although a direct comparison with a standard meta-GA (such as our own) is not given,

papers such as this illustrate how this principle can be extended from its basic form.

[Law and Szeto, 2007]

In this paper, [Law and Szeto, 2007] develop an adaptive GA where parameter adaptation is based on matrices, in particular a mutation matrix and a crossover matrix. This work extends the Mutation Only Genetic Algorithm (MOGA) [Zhang and Szeto, 2005], in which only the mutation rate is adapted.

The novelty of this approach lies in the use of locus statistics, in addition to chromosome fitness, for adapting parameters. In the original MOGA system, a $N \times L$ matrix is constructed for a population of size N containing chromosomes consisting of L loci. Each element in this matrix, which is updated after every generation, represents the mutation probability for the corresponding locus, and is determined according to fitness rankings and loci statistics.

[Law and Szeto, 2007] extends this concept to the crossover operator, by adding a crossover matrix. In this case, the elements in the matrix are based on the Hamming distances between individuals.

Two different variations of this were tested, Long Hamming Distance Crossover (LHDC), and Short Hamming Distance Crossover (SHDC). The authors performed tests using the One-Dimensional Ising Spin Glass benchmark problem, and found that LHDC outperforms SHDC as well as the original MOGA, which in turn has been found to have superior performance for this class of problem than other algorithms.

[Law and Szeto, 2007] presents an alternative way for tackling the GA adaptation problem than traditional self-adaptation or meta-adaptation. The improved performance of this system comes at the cost of an additional statistical heuristic.

[Yun and Gen, 2003]

[Yun and Gen, 2003] propose an adaptive GA based on Fuzzy Logic Controllers (FLCs), and contrast its performance against three other adaptive GAs based on heuristics, as well as a fixed-parameter, canonical GA.

The three heuristic-based algorithms work by adjusting the mutation and crossover rates according to a set of rules, which depend on conditions such as fitness gain/loss across generations, getting stuck in local optima, and relative operator rates between parents and offspring.

The FLC-based algorithm employs two FLCs, one for the mutation rate and one for

the crossover rate (the remaining GA parameters do not adapt). They are both based on the principle of considering the gain or loss in fitness between successive generations, and increasing or decreasing the operator rates according to the corresponding fuzzy decision table.

[Yun and Gen, 2003] performed tests using the Binary, Rosenbrock and Rastrigin benchmark problems. The analysis of their results show that the adaptive algorithms outperform the canonical GA in most cases, with the FLC-based adaptive algorithm showing considerably better performance in search speed and quality than the heuristic-based ones.

[Yun and Gen, 2003] has been included in this review as an example of how other machine learning techniques, in this case fuzzy logic systems, can be used to optimise EAs.

[Zhang et al., 2007]

In [Zhang et al., 2007], the authors propose an adaptive GA capable of adapting its mutation and crossover rate, based on clustering.

[Zhang et al., 2007] divide the optimisation process into four stages: “Initial”, “Sub-maturing”, “Maturing” and “Matured”. In order to determine the stage in which the GA is currently in, they use the K-means algorithm to cluster the distribution of the GA population in the search space, and then consider the relative sizes of the clusters containing the best and worst individual chromosome in the population.

Following this, they use a fuzzy system that, depending on the current stage and based on a set of heuristic rules, increase or reduce the operator rates accordingly.

In addition to a number of benchmarks, [Zhang et al., 2007] tested their system using a real-world application. They optimised a buck regulator (a step-down DC to DC converter) that requires satisfying several static as well as dynamic operational requirements. Their results showed an improvement in performance when compared to buck regulator designs obtained using a traditional GA, which they attribute to their algorithm’s ability to escape local optima.

Like [Yun and Gen, 2003], this paper demonstrates the combination of disparate machine learning techniques (in this case, a cascade of clustering on top of fuzzy logic on top of a GA) to tackle self-adaptation in optimisation problems.

2.5.2 Trust and Reputation Mechanisms

[Gómez Mármol et al., 2011]

In [Gómez Mármol et al., 2011], the authors employ an Ant Colony System (ACS) in order to allow a client *C* to locate the optimal server *S* offering the service *s* within a network. This ACS has been extended in two ways: First, a trust mechanism is added in order to adjust the ACS pheromone routes according to the quality of the service *s* obtained [Marmol et al., 2009]. Further, a GA is employed to optimise the resulting Trust Ant Colony System (TACS) parameters themselves.

The trust mechanism in TACS works by determining the quality of the service *s* obtained by the ACS and, if found lacking, having the client *C* punish the server *S* by evaporating the pheromones leading from *C* to *S*. This is repeated for several iterations, until an optimal service *s* is found.

The introduction of the GA in the system, leading to META-TACS, improves the performance of the TACS algorithm by finding a good set of parameter values for the latter. [Gómez Mármol et al., 2011] used the "Cross generational elitist selection, Heterogeneous recombination, Cataclysmic mutation" (CHC) GA, and performed parameter tuning on the TACS - i.e. the GA was executed before, and not in parallel, with the TACS optimisation.

Although there is a lack of results regarding the actual improvement in performance gained by using the CHC GA, as opposed to a version tuned by hand or otherwise, the authors conclude that their investigation demonstrates the robustness of the trust-based TACS model against a wide range of working parameter values.

As is the case with our adaptive GA, [Gómez Mármol et al., 2011] too use a cascaded machine learning technique, with an "outer" GA optimising the "inner" optimiser - in this case, the trust-based ACS.

[Sutcliffe and Wang, 2012]

[Sutcliffe and Wang, 2012] present a computational model for the development of social relationships, based on Dunbar's Social Brain Hypothesis (SBH) [Dunbar, 1998]. Dunbar's hypothesis rests on a cross-species comparison using behaviour and palaeontological evidence to hypothesise on how complex social structures of friendships arose in social mammals, primates and man.

[Sutcliffe and Wang, 2012] performed a series of experiments in order to investigate how different levels of intimacy in their computational agent model can approach

SBH. In their model, there exist three such levels: strong, medium and weak ties. From the results obtained, the authors conclude that social interaction strategies which favour interacting with existing strong ties or a time variant strategy produced more SBH conformant results than strategies favour more weaker relationships.

For the trust model of [Sutcliffe and Wang, 2012], the authors experimented with both linear and logarithmic functions for trust increase and decay, including all four possible combinations. A result of particular interest was that logarithmic increase and linear decrease produced the best results (closest to SBH), while the other three combinations produced either unstable results, or resulted in few relationships and a wider deviation from SBH.

Although the work in [Sutcliffe and Wang, 2012] concerns social modelling more than it does optimisation, it relates to our work as an interesting study on the effect of different trust adjustment functions.

[Paolucci and Conte, 2009]

In this paper, [Paolucci and Conte, 2009] investigates the extent in which reputation, apart from its role as a deterrent of socially undesirable behaviours (cheating) in multi-agent systems, can also act as an evolutionary drive towards desirable ones.

In previous work [Conte and Paolucci, 2002], the authors found that a minimal set of conditions for the normative population to overcome in efficiency of the cheating population is composed by:

- Memory of past interactions,
- Punishment of cheaters (by cheating them in return),
- Spreading of truthful reputation.

The spreading of truthful reputation is accomplished by having two meeting norm-abiding agents exchange their memory of past interactions, resulting in both agents having a superlist containing both memories.

The authors found that, with information spreading, the average efficiency of the norm-abiding agents was superior. They attribute this to the fact that reputation transmission is less costly than other actions, including moving around - i.e. reputation travels faster than agents, and hence precedes direct experience.

In [Paolucci and Conte, 2009] the authors extend their investigation by introducing information transmission inaccuracies, or noise, into their system. There are two

sources of noise: Copying errors (information noise), and a tendency to forget received information (memory effect).

They performed experiments using two different strategies: One based on courtesy (social optimism), where agents adjust their records according to the ones received by their peers, and one based on calumny (social cynicism), where agents weigh bad received reports more than good ones. The results they present show that, apart from the fact that accurate information is always to be preferred, calumny is preferable both over optimism and over no propagation of reputation.

The authors conclude that current treatments of reputation (e.g., the game-theoretic one) underestimate the role of transmission, and emphasise the importance of accurate information in repeated exchange. They go on to claim that, due to transmission, reputation plays a role not only in repeated encounters, to discourage contract violation, but also in preventing interaction with ill-reputed agents. This, in turn, allows reputation to be characterised as an evolutionary process, characterised by efficient transmission (descent), quite stable even if contradicted by experience (with limited variation), and under some hypothesis endowed with differential survival.

Although fundamentally different in content and scope than our work, [Paolucci and Conte, 2009] is one of the very few works in the literature that relates reputation to evolution - even though its focus is mainly on the transmission of reputation rather than its efficacy as a fitness indicator.

[Hübner et al., 2008]

[Hübner et al., 2008] introduce the notion of "reputation artifacts" whose purpose is to publish objective evaluations of the performance of the agents with respect to their behaviour within the organisation. These evaluations are then retrieved by the members of the organisation in order to build up their reputation of others. These evaluations do not correspond to an agent's reputation directly; instead, they are the means by which reputation is influenced.

Three basic criteria are considered when evaluating an agent within its organisation: *Obedience*, which is computed by the number of obligated goals an agent achieves; *pro-activeness*, which is the number of non-obligatory goals achieved by the agent; and finally *results*, which is a count of the successful executions of schemes wherein that agent participates, regardless of the achievement of the goals in the scheme.

The reputation model proposed by [Hübner et al., 2008] differs from traditional reputation models (such as eBay's, or ours) in that the evaluations are performed by the

infrastructure rather than the individual, and hence the correctness and objectiveness of the information can be assumed. Another difference is that evaluations are not based only on norm conformity, but instead the pro-activeness of the agents is also taken into account. There is a similarity, however, in that the reputation information is published and stored centrally.

[Serrano et al., 2012]

In this paper, the authors propose a method for assessing the reputation of agents using a variety of different metrics, arising from complex interactions (conversation models) that result from structure-rich agent communication languages (ACLs).

The authors support that traditional reputation models, in which an agent's reputation depends solely on a simple interaction success/failure count, do not take advantage of the rich information implicit in MAS-specific ACLs. They continue to suggest that such languages and protocols attempt to capture shared meaning for messages exchanged in MASs, and that the structure and knowledge-level assumptions captured in ACLs and interaction protocols is semantically rich and can be used to extract qualitative properties of observed conversations among agents.

Through experiments in an example e-commerce scenario, they show that their reputation system is capable of effectively utilising the additional information provided by rich interaction protocols and ACLs, and results both in better predictions of future interaction behaviour of evaluated agents, and in improved responsiveness to unexpected changes in others' behaviours.

Although it would have been possible to adopt a similar approach for our reputation system (LCC essentially being a structured ACL), we chose not to do so as such inference techniques would complicate our investigation unnecessarily. However, it is interesting to note that such techniques can be used to improve the performance and robustness of a reputation mechanism.

[Huynh et al., 2006]

In this paper, [Huynh et al., 2006] present the "FIRE" reputation model for open multi-agent systems.

The authors claim that the reliability of reputation models can suffer as a result of a multi-agent system's openness, where various unforeseen changes can occur in the environment at any time. To this end, they propose a system which aggregates a

number of different information sources in order to asses an agent's reputation metric.

The information sources considered are:

- *Interaction trust*, which represents direct (self) experience from past interactions
- *Role-based trust*, which is derived by the various role-based relationships between the agents
- *Witness reputation*, which results from peer reports on an agent's behaviour
- *Certified reputation*, which is derived from third-party references and credentials provided by the agent itself.

Agents are free to use any combination of these sources, however the experiments carried out by the authors demonstrate that the highest level of precision is obtained by combining all four of them. Further, a comparison of this decentralised system with the "Sporas" model [Zacharia and Maes, 2000], a centralised one, resulted in similar performance.

Our own reputation system can be regarded as a subset of "FIRE", in that it incorporates *Interaction trust* and *Witness reputation*, and is similarly decentralised. "FIRE" takes this further by incorporating additional reputation sources, namely inference (*Role-based trust*), and a third-party authority (Certified reputation).

[Burnett et al., 2011]

In [Burnett et al., 2011], the authors claim that although trust is crucial in dynamic, open multi-agent systems, where agents may join and leave at any time, it is not enough as it may be difficult for agents to form relationships that are stable enough for confident interactions. They propose a decision-theoretic model of trust decision making that allows further controls to be used in addition to trust, leading to an increase in confidence during initial interactions.

The weakness of traditional trust models identified by the authors of this paper lies in the fact that, in order for an agent to build a reliable measure of trust for its peers, a base of interactions are required from which to form generalisations. They continue to claim that, in the case of open MASs, this limitation becomes even more problematic, due to the high population turnover of such systems.

The architecture proposed by the authors is based on additional controls that allows agents to delegate the perceived risk of initial interactions. They consider three such controls:

- *Explicit incentives*, by which the trustor creates a contract with the trustee specifying the compensation that the latter will receive depending on the outcome of the interaction.

- *Monitoring*, by which the trustor expends additional effort in order to observe the behavioural choices of the trustee.

- *Reputational incentives*, by which the trustor calculates the reputational gain/loss that the trustee will receive, as an additional incentive.

The authors show that, by employing controls in addition to trust, trustors can mitigate some of the perceived risk in their interactions, and be motivated to delegate, providing crucial initial interactions required to bootstrap trust. The experiments they conducted demonstrate that in certain circumstances, decision-making and delegation using a mixture of trust and control can be beneficial, even when those controls are costly to implement.

The findings as well as the proposed architecture in [Burnett et al., 2011] would be very relevant in a real-world application of our system, where - as an open P2P platform - its performance would inevitably suffer from a high population turnover.

[Hazard and Singh, 2010]

In this paper, [Hazard and Singh, 2010] address the commonalities between trust and reputation systems as two distinct approaches, and connect the two architecturally and functionally. They present a life cycle model for such systems, and investigate the effect that *signalling* and *sanctioning* have on a given system.

Signalling and sanctioning models originate in game theory. In the former, agents attempt to assess private attributes of other agents, while in the latter, agents behave strategically in order to maximise their own utility. [Hazard and Singh, 2010] present a heuristic that allows them to determine how a system is governed between signalling and sanctioning, which is prescriptive in terms of what kind of a reputation or trust model should be used for a given situation. By measuring the effects of signalling and sanctioning, [Hazard and Singh, 2010] attempt to unify reputation systems, trust systems and related game theory under a common architectural framework.

Their practical investigation, based around an online auction model, demonstrates that reputation systems can benefit from putting different emphasis on signalling and

sanctioning, depending on the type of attack (identity drop, sybil attack, defection etc.) that is being addressed.

This approach would benefit our own trust/reputation mechanism as well, especially if it had to cope with more types of attack than just defection.

[Wang et al., 2011]

In [Wang et al., 2011], the authors address the problem of *maintaining* and updating trust as opposed to acquiring and sharing it. The model they propose is deliberately probabilistic rather than heuristic.

[Wang et al., 2011] consider probability and certainty as two dimensions of trust. In their model, trust updates are based on two modes:

- *Trust update for referrers*, wherein an agent updates the trust it places in a referrer based on how accurate its referrals are. This is a way for an agent to maintain its social relationship with a referrer.

- *Trust update for history*, wherein an agent updates the trust it places in a service provider by tuning the relative weight assigned to the service providers past behaviour with respect to its current behaviour. This is a way for an agent to accommodate the dynamism of a service provider.

[Wang et al., 2011] have tried various update mechanisms for these two modes, including linear, averaged, as well as probabilistic ones (based on probability and certainty metrics).

The authors conducted experiments, in which they also included malicious referrer agents - who provide trust reports indicating a falsely exaggerated amount of evidence. The results they obtained show that the proposed model provides accurate estimates of the trustworthiness of agents that change behaviour frequently, and is also able to capture the dynamic behaviour of the agents.

Like us, [Wang et al., 2011] avoid using heuristic rules, and instead rely on a purely probabilistic model of trust. The most significant difference between our system and the one presented in [Wang et al., 2011], is that the latter introduces an additional level of trust, so that trust for a referrer is decoupled from trust for a service provider (the two being identical in our case). In addition, defection in [Wang et al., 2011] is manifested as erroneous referrals, whereas in our case, it is the service that is erroneous, with referrals being always true (although they remain arbitrarily misleading when provided

by defective agents, as the latter lack the context required to distinguish between good and bad peers). Despite this, work such as [Wang et al., 2011] demonstrate how a basic reputation mechanism, such as the one we use in our own system, can be modified in order to improve its efficiency, according to the conditions in which it needs to perform.

[Zacharia and Maes, 2000]

This paper presents "Sporas" and "Histos", two of the most frequently encountered reputation models in the literature.

[Zacharia and Maes, 2000] focus primarily on trust as this relates to online communities, such as the web, and in particular e-commerce services such as eBay and Amazon. They claim that the existing reputation models employed by such services suffer from scalability issues, namely the fact that due to the scale of such rating systems, users are reluctant to give low scores to their trading partners, which reduces the value of the rating system.

"Sporas" addresses these issues by adding a number of rules. In particular, it is based on the following principles:

- New users start with a minimum reputation value, and they build up reputation during their activity on the system.
- The reputation value of a user does not fall below the reputation of a new user no matter how unreliable the user is.
- After each rating, the reputation value of the user is updated based on the feedback provided by the other party to reflect his/her trustworthiness in the latest transaction.
- Two users may rate each other only once. If two users happen to interact more than once, the system keeps the most recently submitted rating. That way they avoid artificially inflated reputations through two-party collusions.

"Histos" extends "Sporas" in that it offers a more personalised metric of reputation for each participating agent. In "Sporas", the reputation value for each individual is global and common for all its peers. In "Histos", the reputation of each agent differs for each one of its peers, according to past interactions that took place between them, directly or indirectly through the resulting social network.

This paper has a direct parallel with our investigation of different reputation mechanisms (Section 6.2.2), as it underlines the difference between a centralised reputation system ("Sporas", cf. our "central" reputation model) and a personalised one ("Histos", cf. our "memory" and "collective" reputation models).

[Teacy et al., 2006]

[Teacy et al., 2006] present the "Trust and Reputation model for Agent-based Virtual OrganisationS" (TRAVOS) model, another popular reputation mechanism.

With "TRAVOS", the authors attempt to address the problem of untrustworthy and/or inexperienced agents within large-scale multi-agent systems, by combining a trust mechanism with a reputation mechanism (where there is a lack of first-hand experience) that draws upon experience from third parties. The two are combined using a Bayesian framework.

Of particular importance is the assessment of the trustworthiness of reputation, or opinion, providers. [Teacy et al., 2006] identify two methods for dealing with this latter requirement: *Endogenous*, in which unreliable reputation information is identified by considering the statistical properties of the reported opinions alone; and *exogenous*, which relies on other information to make such judgements, such as the reputation of the source or its relationship with the trustee. In "TRAVOS", the authors follow the latter approach, in that they judge a reputation provider on the perceived accuracy of its past opinions, rather than its deviation from mainstream opinion.

The authors performed experiments using a simulated marketplace environment, and compared the performance of "TRAVOS" to another reputation model, the "Beta Reputation System" (BRS) [Ismail and Josang, 2002], which is based on the beta distribution. The results they present demonstrate that "TRAVOS" performed significantly better than the BRS benchmark. Additionally, "TRAVOS" was able to extract a positive influence on performance from reputation, even when 50% of sources were intentionally misleading. Finally, their empirical evaluation showed that when 100% of sources were misleading, reputation had a negative effect on performance.

As in [Wang et al., 2011], [Teacy et al., 2006] makes a distinction between trust for a service provider, and trust for an opinion provider. Although in the current implementation of our system that distinction does not apply, this paper makes an interesting case of using inference, particularly on the relationship between suggesters and trustees, in order to come up with a more reliable trust measure for the former.

[Sabater and Sierra, 2001]

In [Sabater and Sierra, 2001], the authors present "REGRET", a well-established reputation model that takes into account the social dimension of the participating agents, as well as a hierarchical ontology structure. These two characteristics allow "REGRET" to consider several types of reputation at the same time.

The ontological approach taken by [Sabater and Sierra, 2001] is based on the premise that reputation is compositional, which means that the overall opinion on an entity is obtained as a result of the combination of different pieces of information - i.e. a good reputation for service *seller* is a function of the reputations for sub-services *delivery*, *price* and *quality*, where all three sub-services contribute to service *seller*.

Each individual in the "REGRET" system has a distinct ontological structure that allows it to combine these different types of reputation using different weights. The authors term the different types of reputation and the way these are combined, the *ontological dimension* of reputation. This in turn is combined with the *individual dimension* (individual trust) and the *social dimension* (collective reputation), in order to obtain a final measure of an agent's total reputation.

The authors compared the performance of "REGRET" against the "Sporas" model [Zacharia and Maes, 2000], as well as the method used in Amazon Auctions. Their results show that the added ontological dimension allows "REGRET" to be used successfully in agent societies with different structures.

It would be possible to adopt this approach in our own system, provided we could come up with a meaningful ontology for the service provided by the agents. For example, a particular agent may generally result in a fitness gain (good *quality*), but is slow to answer (bad *delivery*) - which reduces the overall conferred benefit. Added complexity and ontology heuristics/analysis aside, such an approach would probably prove to be very powerful in a system with multiple classes of solver, as it would provide a context for balancing different service elements between non-uniform agents. However, due to the uniformity of the agents in the current implementation of our system, we would expect the benefits of the "REGRET" approach to not be particularly pronounced.

2.6 Summary

In this chapter, we have discussed the following points:

- A brief introduction to evolutionary algorithms, and how these can be used to facilitate a number of large scale applications.
- A description and comparison of commonly encountered evolutionary algorithms (EA), namely Genetic Algorithms (GA), Evolution Strategies (ES), Evolutionary Programming (EP) and Swarm Intelligence (SI).
- The problem of parameterisation in GAs, and common approaches to adaptation that can be used to alleviate it.
- The issue of parallelisation in GAs, and the three general architectures commonly used to address it.
- The driving ideas behind distributed computing, as well as a comparative description of grid, peer-to-peer (P2P) and multi-agent systems (MAS), three common types of distributed systems.
- The particular properties and requirements for communication standards and security pertaining to open distributed systems.
- An introduction to the concept of electronic institutions and the Lightweight Coordination Calculus (LCC), which was used as the communication medium in our system.
- A discussion on the need for trust and reputation mechanisms in open systems, including their definition and description.
- A literature review of the state-of-the-art work related to our research.

Chapter 3

Platform Implementation

3.1 Overview

The first step towards the implementation and testing of our proposed theory was ensuring that we have an adequate, reliable platform at our disposal in which to design our algorithms and conduct our experiments.

In this chapter we discuss various technical aspects of our platform, the LiJ interpreter, and present a broad overview of its architecture and inner mechanics. In addition, we discuss a number of non-trivial problems that we encountered, along with the solutions that enabled us to produce a working, robust software platform on which to base our research.

The LiJ interpreter, including the latest binaries, the full source code and a number of examples, can be found at the interpreter's homepage on SourceForge, at `http://sourceforge.net/projects/lij/`.

3.2 Choosing Tools

3.2.1 Requirements

The first step in the implementation of our platform involved identifying the requirements that it must satisfy. In particular, our system must:

- Be suitable for developing and deploying multi-agent systems, with intrinsic support for inter-agent communication, coordination, and execution of clauses.

- Be deployable in distributed computation environments, such as a grid computation network, a cluster, or a multi-processor computer.
- Allow for reusability of components and agent specifications, so as to minimise the coding effort required for testing multiple different algorithms.
- Be platform- and OS-independent, so as to be able to take advantage of all the computational resources available to us.
- Be reliable and robust, capable of decent performance in order to accommodate lengthy, large-scale experiments with acceptable efficiency.

3.2.2 Existing Platforms

As parallel computation platforms become more mainstream (e.g. cloud computing, grid computing, multi-core processors etc), multi-agent development tools become more abundant and more mature. What follows is a concise survey of some of them. Further details and more extended comparisons are given in surveys such as [Ricordel and Demazeau, 2000] and [Allan, 2009].

AgentBuilder

AgentBuilder is an integrated software toolkit for rapid development of intelligent agents. It is based on the BDI model, which lends it a methodology with a solid academic background.

AgentBuilder, which is based on Java, provides for autonomous agent operation, networking and agent communication. It provides graphical tools for specifying the behaviour of agents, as well as debugging and monitoring their operation, and analysing the problem domain.

AgentBuilder is closed-source software, and is available for a fee.

Jack

The Jack multi-agent environment is also Java-based, and also grounded on a BDI model. Agent behaviour is specified using an extension of the Java language, called the Jack Agent Language, which is later translated into standard Java by means of the Jack Agent Compiler. Like AgentBuilder, it provides a graphical tool for project management, although it lacks tools for problem analysis.

Jack is also closed-source and available for a fee.

MadKit

The MadKit platform is similarly based on Java, although unlike the previous two, it comprises principally a multi-agent runtime engine rather than a development environment. It does, however, include graphical tools for deploying and monitoring multi-agent systems. MadKit agent operation follows the Aalaadin model, which is based on agent organisation, interaction protocols, and agent tasks and goals

MadKit is open-source software still under active development, and available for free under the GNU GPL license.

Zeus

Of all the multi-agent platforms we have discussed so far, Zeus is arguably the most complete one. It puts a strong emphasis on methodology (the Zeus methodology), and offers tools for all stages in it: analysis (via UML diagrams), design, development and deployment of multi-agent systems.

Zeus is open-source software, distributed under the Mozilla license.

Jade

Jade is another multi-agent platform based on Java, where agents are Java classes themselves (extending the "Agent" class). Agents can define any number of behaviours (by inheriting the "Behaviour" class or its subclasses), and pass FIPA-compliant messages among them as standard Java objects.The Jade runtime handles communications transparently, using local method invocation, RMI and/or TCP/IP as necessary.

Jade itself is written in Java, and is available as open-source software.

Jason

Jason allows the development of multi-agent systems wherein agents use an extension of the AgentSpeak BDI programming language in order to specify their behaviour. It lacks an IDE for development, and instead comes as a plugin for Eclipse/gEdit.

Jason is open-source software written in Java, and is offered free-of-charge under the GNU LGPL license.

3.2.3 LiJ and the OpenKnowledge Framework

Even though the OpenKnowledge kernel provides intrinsic support for the execution of distributed LCC protocols, our first attempts to use it for our experiments were met with frustration. The reason was that the system's infrastructure was simply too large to allow us to work efficiently.

Far from being just a proof-of-concept, the OpenKnowledge framework was designed for actual network deployment. This involves the use of a discovery service (similar to a Java RMI registry in scope but with the additional functionality of protocol registration and discovery), message passing via TCP/IP with all the associated costs incurred by flow control, object serialization etc, support for agnostic registration of agents (as we would also require in a real-world deployment scenario for our algorithms), support for ontology mapping that allows agents of diverse origin to understand each other, and a lot of additional functionality that we simply did not need to test our algorithms. As a result, simple experiments required inordinate amounts of time to deploy, and more so to automate.

The solution to this problem came in the form of LiJ, the LCC Interpreter for Java. LiJ is a from-scratch reimplementation of the interpreter component in the OpenKnowledge kernel, but stripped of all the functionality that was unnecessary for our purposes: LiJ is single-machine but multi-threaded (with each agent being executed in its own individual thread), it involves no networking or discovery service (although these can be added in the future in the existing design), and in short provides a barebones LCC interpreter, extremely robust and reliable, and many times faster in a single-machine or CPU cluster environment (due to the lack of additional overhead).

In the following section, we provide a concise analysis of the LiJ interpreter.

3.3 The LiJ Interpreter

3.3.1 Class Structure

Our experimentation software platform consists of two main components:

- The LCC Interpreter, LiJ.
- The Java classes that provide the constraint method implementations for the agents in our experiments, as well as a basic GUI for monitoring the progress of each agent.

In this section we present a very brief overview of the software architecture for LiJ, by means of UML class diagrams. We do not discuss the agent-specific classes, the workings of which are more-or-less trivial. However, we have included the source code for these in Appendix C for reference.

The intention is not to present a full technical analysis of the interpreter; rather, we aim to illustrate its main components and the relationship between them, as well as the way in which the various semantic components of LCC (Def's, constraint types etc) are mapped into it.

The LiJ interpreter comprises eight packages:

- *lij.exceptions*

  This contains the Java exception classes that are used throughout LiJ.

- *lij.interfaces*

  A small number of interfaces that must be implemented by external/third-party agent software, in order to be able to participate in LCC interactions run under LiJ.

- *lij.model*

  Each of the classes in this package represents a construct of the LCC definition (see Appendix A). They are created during the parsing of the LCC protocol, and provide a programmatical representation of it.

- *lij.monitor*

  This package contains classes that implement an (optional) monitor GUI for LiJ, capable of providing feedback about the current state of the subscribed agents, the contents of the message buffer, as well as a message log.

- *lij.parser*

  The classes in this package are generated exclusively by the parser, *javacc*, which is discussed in more detail in the next section.

- *lij.parserutil*

  The methods of the single class in this package, *TreeFactory*, are called during parsing by *javacc* and aid in the creation of a tree representation of the LCC protocol. This process is discussed in more detail in Section 3.3.3 below.

- *lij.runtime*

  This is the main package of the interpreter. It contains the main class (*Interpreter*), in which an LCC protocol source is initially loaded for execution. In addition, it contains code for other aspects of code execution, such as the message buffer, handling of the symbol table etc.

- *lij.util*

  This package contains various utility/helper methods.

Figures 3.1 and 3.2 show UML class diagrams for the two principal packages of LiJ, *lij.model* and *lij.runtime*, respectively. It must be noted at this point that these diagrams are not complete: Some relationships are not shown, some minor classes have been omitted, and the list of attributes and operators are restricted to the most important members. This was done in order to help focus on the overall structure of the design, rather than providing complete diagrams at the cost of reduced readability.

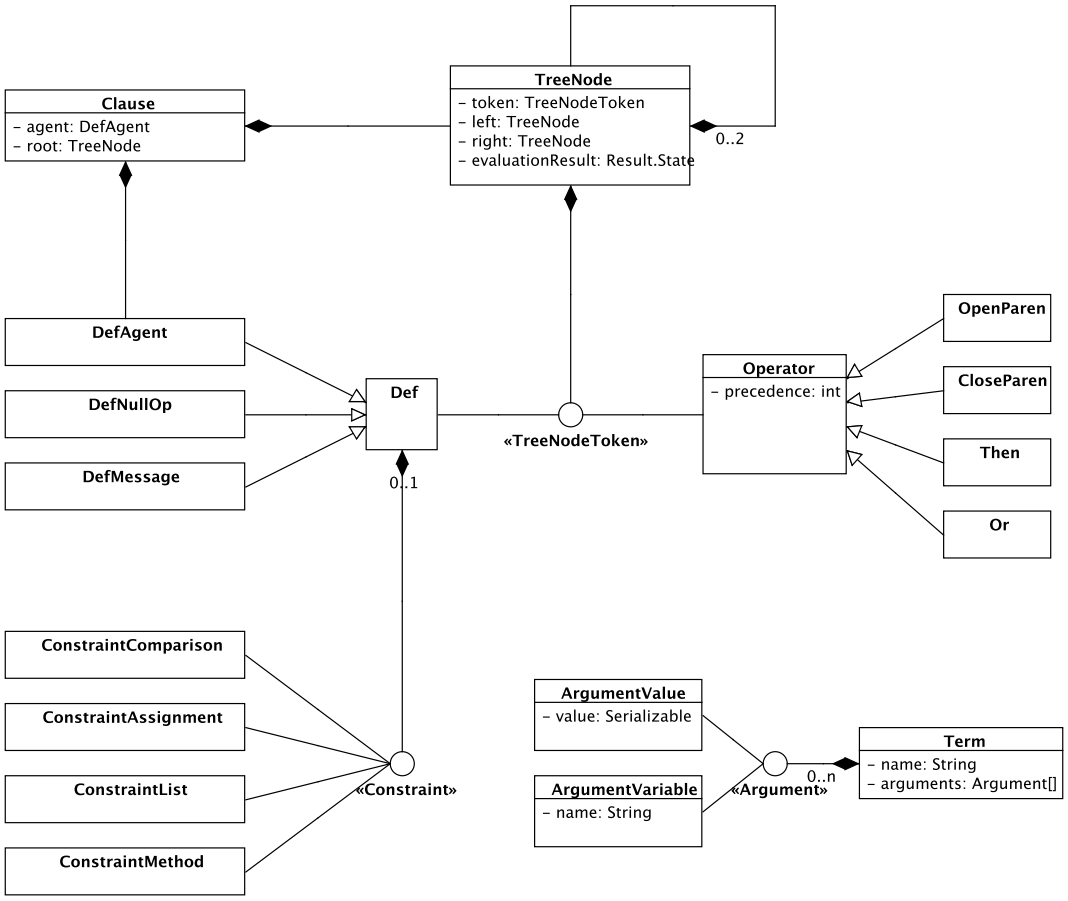


Figure 3.1: Concise UML class diagram for the *lij.model* package of the LiJ interpreter.

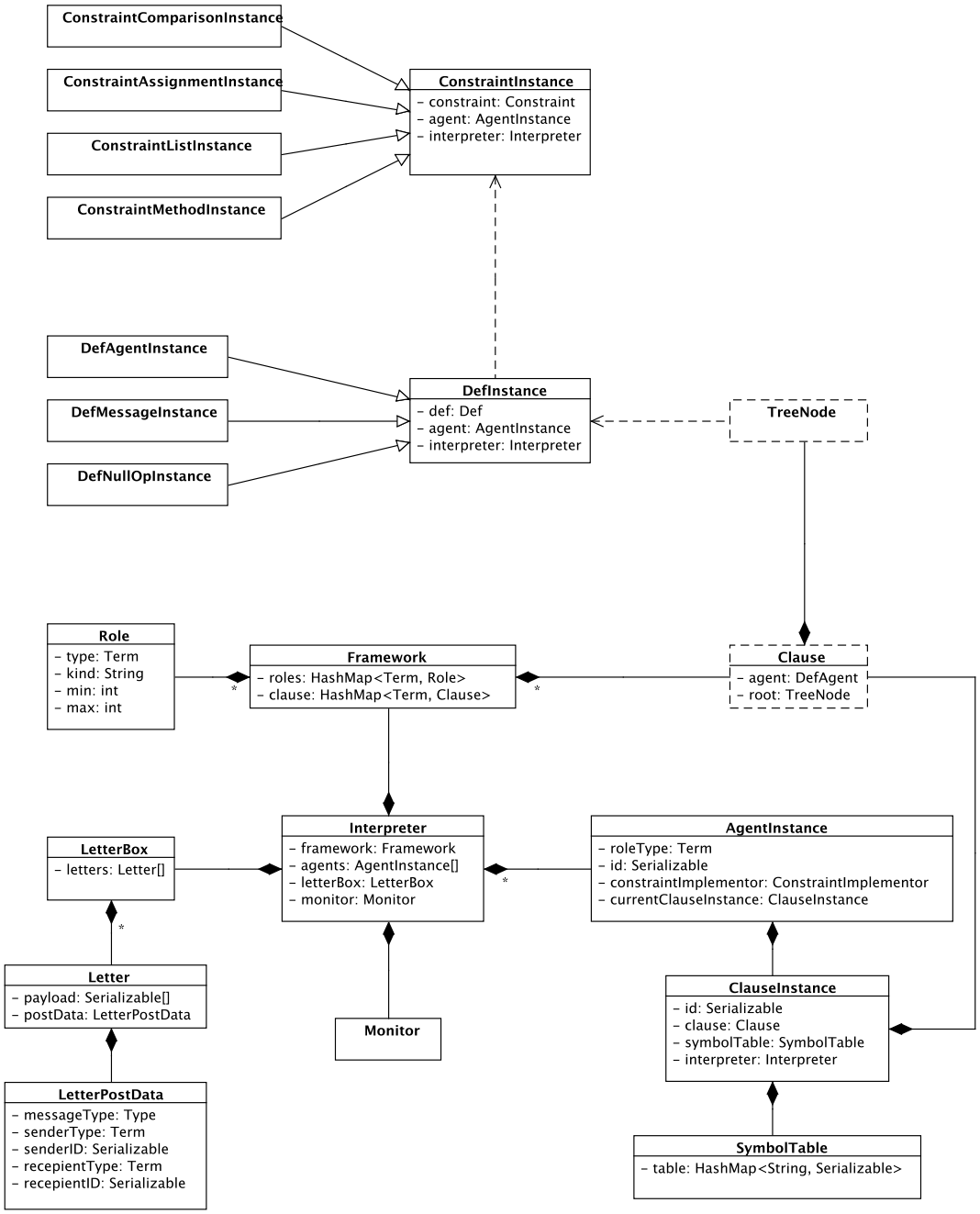


Figure 3.2: Concise UML class diagram for the *lij.runtime* package of the LiJ interpreter. Classes with dashed outlines are part of the *lij.model* package.

3.3.2 Parser

LCC protocols are described in LCC source files, in ASCII text format, which must then be parsed by the interpreter in order to be executed. The task of parsing these text files is not trivial; it involves recognising the various tokens that comprise the LCC protocol, as well as enforcing grammatical and syntactical rules and checking for errors.

In order to tackle this aspect of the software, we chose to use *javacc*, the "Java Compiler Compiler" parser generator. In order to work, *javacc* needs a syntax definition file appropriate for the particular language being parsed. This file contains the lexical semantics of the language, i.e. a description of all the acceptable tokens that comprise a source file, as well as the logical order and structure in which these tokens can be arranged.

By providing it with an appropriate LCC syntax definition file, *javacc* is able to generate the Java classes that comprise the parser, which can then be incorporated into the LiJ interpreter. Upon encountering valid tokens, the parser code can then call appropriate methods in our interpreter code that dictate how these tokens will be handled.

3.3.3 Tree Generation

Given the LCC syntax definition file, the javacc parser is perfectly capable of handling the token parsing from the LCC source file without us needing to do anything more. However, as a generic parser, it is not able to construct on its own a usable, programmatically working model of the LCC protocol from the tokens it parses. In order to obtain such a model, we follow these steps:

1. We instruct the parser to add the tokens it parses (operators and Def's) into a list, in the order they are encountered in the LCC protocol source file.

2. We rearrange the tokens and transform the LCC protocol into its postfix / Reverse Polish Notation (RPN) form.

3. We use the RPN-arranged tokens to construct a tree, wherein branch nodes represent operators (*THEN*, *OR*), and leaf nodes represent Def's.

The way RPN reordering works is the following: We start with a fully populated list containing all tokens in their original order, a temporary stack, as well as an empty

"RPN" list wherein we will put the tokens in RPN order. Then, we consider each token in the original list in turn:

1. If a Def token is encountered, we add it in the RPN list.
2. If an operator token is encountered, we compare its precedence value with that of the operator on the top of the stack:
   (a) If it is found higher, we push the current operator on the stack.
   (b) If it found lower, we pop the top operator from the stack into the RPN list, then reapply step 2.
3. If an open-parenthesis is encountered, we push it on the stack.
4. If a close-parenthesis is encountered, we pop everything from the stack into the RPN list, until an open-parenthesis is popped from the stack.
5. If we have reached the end of the original list, we pop all remaining tokens from the stack into the RPN list (we can consider this to be the "EOF" operator, having the lowest precedence of all).

There are two types of operator (*THEN* and *OR*), and three types of Def (agent, message and the null operator). We have given the *THEN* operator a higher precedence than *OR*, which means that *THEN* operators will be evaluated before *OR* operators (similar to the way that, in standard arithmetic, the $\times$ and $\div$ operators have a higher precedence than the $+$ and $-$ operators.

Having reordered the tokens in RPN order, it is easy to assemble the operator tree as follows: We start with the RPN list obtained above, as well as a temporary stack. We then consider each token in the RPN list in turn:

1. If a Def token is encountered, we push it on the stack.
2. If an operator is encountered, we pop the two topmost tokens from the stack, we create a new operator tree node with these two tokens as its children, and then we push the newly created node on the stack.

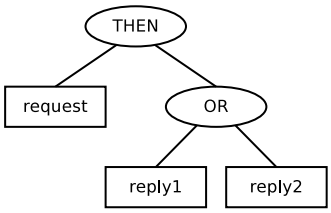
Because of the way RPN works, we can be certain that at the end of this process we will end up with a single tree node in the stack. This tree node is the root node of our tree, which can now be used dynamically in the interpreter for the execution of the LCC protocol. Figure 3.3 illustrates an example of this process.

```
request => a(Type, Id)
then
(
  reply1() <= a(Type, Id)
  or
  reply2() <= a(Type, Id)
)
```

(a) A simple LCC protocol source.

```
request => a(Type, Id)
reply1() <= a(Type, Id)
reply2() <= a(Type, Id)
or
then
```

(b) The same LCC protocol in Reverse Polish Notation



(c) The resulting tree representation of the LCC protocol.

Figure 3.3: Example illustrating the conversion of a simple LCC protocol to its programmatically usable tree form using the RPN process.

3.3.4 Tri-state Logic and the Committed Choice Issue

One of the trickiest problems that we encountered during the implementation of the LiJ interpreter involved the handling of incoming message Def's. The problem stems from the asynchronous nature of message passing in a open, networked environment.

In a logic program based on static knowledge states, it is often possible to evaluate a typical Def to either a *true* or *false* value immediately. With incoming messages, however, things get a bit more complicated, for two reasons: a) Messages may not necessarily arrive in the order they are expected to, and b) a message may take a long time to arrive, or, in other words, an incoming message may not necessarily be in the message buffer when we request it.

Without altering the semantics of the language, there are two ways in which to implement message reception: a) blocking, and b) non-blocking. In the first case, whenever we encounter an incoming message Def, we block the execution of the protocol until the expected message arrives in the buffer. In the second, we treat each yet-unreceived incoming message as a failure to receive, and evaluate the corresponding Def to *false*.

Both these approaches suffer from significant problems. In the case of blocking

reception, we are unable to handle parallel incoming message Def's (separated by *or* operators). As a result, there is no way to implement a clause that is supposed to receive one of, say, three different messages in parallel (with each forking to a different sub-clause), since the interpreter would block (possibly forever) on the first incoming message Def, and would ignore any incoming messages that correspond to the other two.

A non-blocking scheme, on the other hand, is unable to deal with asynchronous message passing altogether, since it may disregard an incoming message (and evaluate the respective Def to *false*) that took a bit too long to arrive.

It is obvious that neither of these options would help us produce a useful interpreter. Hence we had to come up with a third solution: that of tri-state logic (a similar use of tri-state logic in logic programming can be found in [Jaffar et al., 2007]).

In our particular case, the third logic state (in addition to *true* and *false*) is *maybe*. The *maybe* state stands for exactly that: the real, final state of a Def that is evaluated to *maybe* is still uncertain, but we do not have to block at its evaluation. Instead, we can continue down the tree until we have a definitive answer, and if we don't, we re-evaluate that node (and all its still undetermined children) until we do.

Tables 3.1a and 3.1b show the truth tables for the tri-state *then* and *or* operators respectively.

A	B	A ∧ B
T	T	T
T	F	F
T	?	?
F	T	F
F	F	F
F	?	F
?	T	?
?	F	F
?	?	?

(a) *AND* operator

A	B	A ∨ B
T	T	T
T	F	T
T	?	T
F	T	T
F	F	F
F	?	?
?	T	T
?	F	?
?	?	?

(b) *OR* operator

Table 3.1: Truth tables for the tri-state *THEN* (∧) and *OR* (∨) operators.

This scheme translates to message passing as follows: An incoming message Def will be evaluated to *true* only if it is found in the message buffer. If it is not there, the Def will be evaluated to *maybe* - which is to say that, the message is not here right now, but it may arrive in the future. Finally, in the case that an incoming message does not match one of the incoming messages specified at the current point in the protocol, it will simply remain in the message buffer until (and if) it is needed in the future.

The best way to understand this concept is by considering the example illustrated schematically in Figure 3.4.

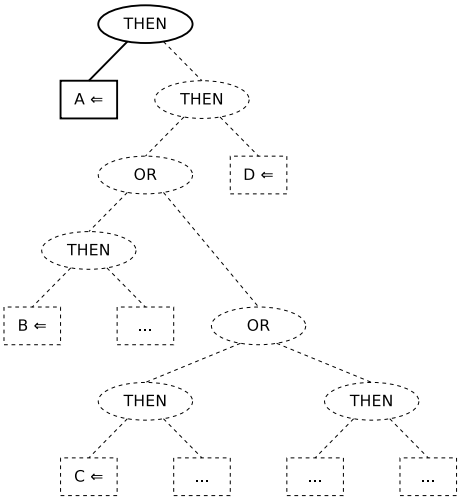
3.3.5 Infinite Recursion and Cyclic Clauses

Another problem that we had to tackle in the interpreter involved cases where clauses need to loop, or recurse, infinitely. Although traditionally loops have not been a principal concern in logic programming language design, we had a very real need for them since in our experiments we were dealing with continuous processes as opposed to one-off evaluations.

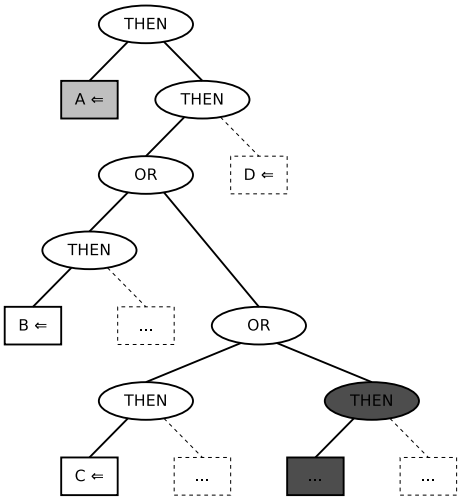
The typical way of handling loops in logic programming languages involves the use of tail recursion, where the very last Def in a tail-recursive clause is a call to the clause itself. This approach works well for small iterative problems, but has the drawback that, in large problems involving long-running loops, the interpreter (itself based on procedural Java) would eventually crash with a stack overflow.

This problem can be solved in two ways: The first, which was deemed too involved for the scope of our investigation, involves using fairly advanced flow analysis of the program being executed for tail-recursion elimination, as described in [Muchnick, 1997]. The second involves the addition of extra, loop-specific constructs in the language itself. Among others, [Schimpf, 2002] discusses a number of ways in which loop constructs can be implemented in Prolog-based constraint logic programming languages, such as LCC.

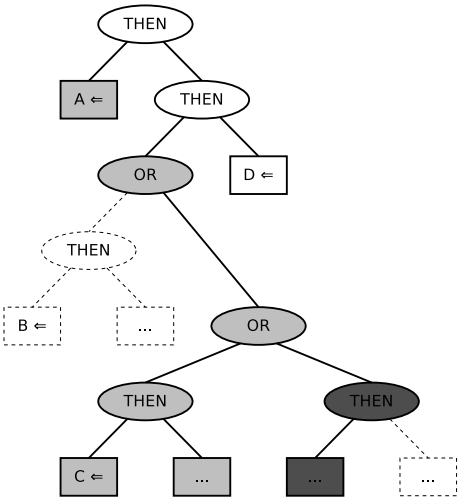
In our case, we chose not to add further primitives into the LCC language; instead, we added a further role type to the list of existing ones (see Appendix A for a description of these). Clauses based on this additional role type, named “cyclic”, are handled differently by the LiJ interpreter, which will automatically repeat the execution of such clauses without the need of a tail-recursion clause Def - although the latter remains an option for simpler cases.



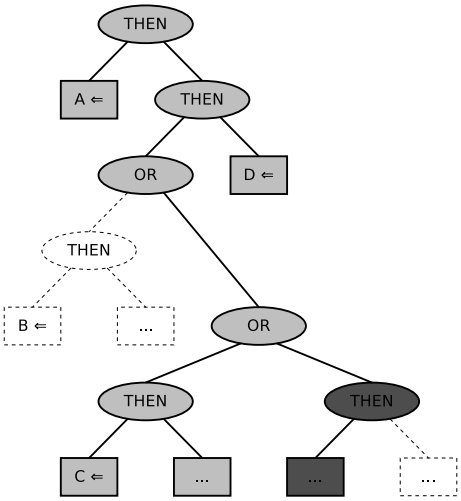
(a) Initially, we are waiting for message A.



(b) Message A arrives, but neither message B nor message C is in the buffer yet.



(c) Message C arrives, so now we wait for message D.



(d) Finally, message D arrives, and the root node result is resolved to TRUE.

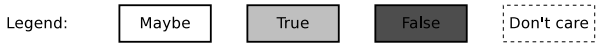


Figure 3.4: Example illustrating the use of tri-state logic in the LiJ interpreter for concurrent message handling.

3.4 Summary

In this chapter, we have discussed the following points:

- The requirements that our experimentation platform needed to satisfy.
- An overview of some popular MAS platforms.
- The reasons behind our choice to implement a custom platform, as opposed to using one of the existing third-party tools.
- A brief overview of the class architecture of LiJ, the interpreter that allowed us to execute our distributed algorithms, expressed in the LCC language.
- The way in which we implemented parsing in LiJ using RPN.
- The way in which we construct a programmatically working tree model of an LCC protocol from the parsed source file.
- The solution of tri-state logic to the issue of committed choice, which is especially important for asynchronous passing of messages between coordinating agents.

Chapter 4

Evaluation Methodology

4.1 Overview

This chapter presents some key issues that relate to our experimentation methodology.

Initially, we discuss how we measure the performance of each algorithm/configuration, as well as how we ground these performance results statistically.

Further, we present the three benchmark functions that served as the optimisation problems in our experiments, along with a brief overview of their particular characteristics and the configuration we used.

Next we explain the behaviour of a special class of "lying", or defective, agent, that we introduced in some of our experiments in order to contrast how different algorithms cope in the presence of noise.

Finally, we provide a brief discussion on the autonomy and motivation of the agents in our system, two points of particular importance for multi-agent systems.

4.2 Measuring Performance

In general, the performance of a GA is measured by the time it takes to achieve the required result - usually convergence to a stable or pre-set fitness. Alternatively, a GA may be allowed to run for a predetermined number of generations (which translates into time), in order to determine the final best fitness. To allow for differentiation between hardware platforms, this time is typically estimated by taking into account the total number of fitness evaluations that occur in the system.

In our experiments, however, we deemed it more appropriate to use the number of generations required for a specified target fitness score as a measure of performance.

The reason for this is that, in our system, we can assume that each agent runs in parallel with its peers, hence the actual time required by one agent is more-or-less equal to the time required by all of them, given a true parallel/distributed computational platform.

In all of our experiments, the intra-agent GA population size was the same, and thus the execution time required for the evaluation of every population (typically the most computationally intensive task in a real-world GA application) was also the same. Therefore, we can assume that the number of generations taken by the algorithm to converge is proportional to the actual time it would require on a benchmark computational system that would allow for parallel execution of the agents.

This, of course, does not take into account the overhead incurred by network communication; however, as this overhead is again more or less equivalent in all experiments, we can safely factor it out.

4.3 Significance of Results

4.3.1 The Mann-Whitney U-test

In order to estimate the significance of our findings, we performed a concise statistical analysis of our results using the Mann-Whitney U-test (also known as the Mann-Whitney-Wilcoxon test, or the two-sample variant of the Wilcoxon test). We chose this particular test as it is non-parametric, and hence able to deal with non-normal data distributions, such as those in our results.

The word *significance* in statistical analyses can mean different things to different people. In order to avoid confusion, as a measure of the significance of our observations we refer to the p-value threshold that was used to reject the null hypothesis (of the two samples under comparison having the same medians).

4.3.2 Number of Runs

In order to compensate for the stochastic nature of the experiments and produce more meaningful results, all runs were executed 40 times and their output was averaged. In this way, we exceed the rule-of-thumb of 30 dictated by the central limit theorem for a sufficiently large number of runs, and ensure the reliability of our results.

In all experiments where algorithms had to reach a certain target, we followed a zero-tolerance policy towards failed attempts. In all cases where one or more runs (out

of 40) failed to reach the target, the results were discarded and the experiment/configuration was deemed failed.

4.4 Benchmark Functions

4.4.1 Rastrigin Function

As our main optimisation test case, we used the Rastrigin equation. Its general form is given in Equation 4.1.

$$F(\bar{x}) = kA + \sum_{i=1}^{k} (x_i^2 - Acos(2\pi x_i)) \tag{4.1}$$

A plot of a simple version of the Rastrigin function, using only two dimensions (i.e. two Rastrigin variables), can be seen in Figure 4.1.

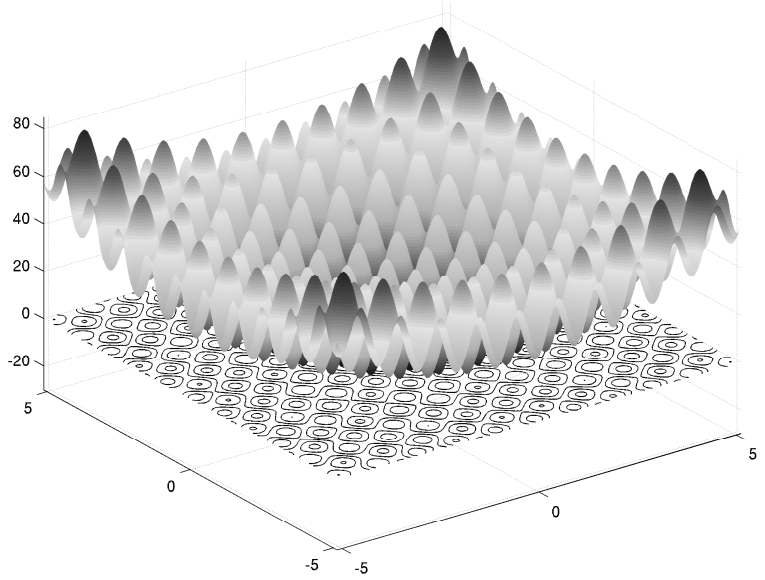


Figure 4.1: Plot of the Rastrigin function for *k=2* variables.

The Rastrigin function is a popular test function widely adopted in the GA field, as it represents a challenging fitness landscape for optimisers. It consists of multiple local minima and maxima, but only a single global minimum. Its multiple valleys provide ample opportunity for a poorly performing GA to get trapped in, while the global minimum (located at the "centre" of the fitness landscape, where all the Rastrigin variables have a value of zero) is typically set as the objective for the optimiser being tested.

This is a minimisation problem, which implies that the aim of the GA is to make the fitness measure as small as possible, with the optimal value being zero.

In all our experiments, the steepness A was set to 10, and the number of Rastrigin variables k was set to 100 - making our fitness function quite challenging compared to similar cases in the literature. The range of x was -0.5 to +0.5, encoded in 16-bit Gray code. The choice of these parameters was influenced by similar experiments in the literature (e.g. [Takashima et al., 2003]).

4.4.2 Sphere Function

In addition to the Rastrigin function, we also performed a number of comparative tests (see section 5.3.5) using two additional benchmark functions, both taken from De Jong's test suite [De Jong, 1975].

The Sphere function is probably the simplest in the set. It does not contain any local optima, but instead only a general global optimum in the centre, with a value of zero.

The general form of the Sphere function is given in equation 4.2.

$$F(\bar{x}) = \sum_{i=1}^{k} x_i^2 \tag{4.2}$$

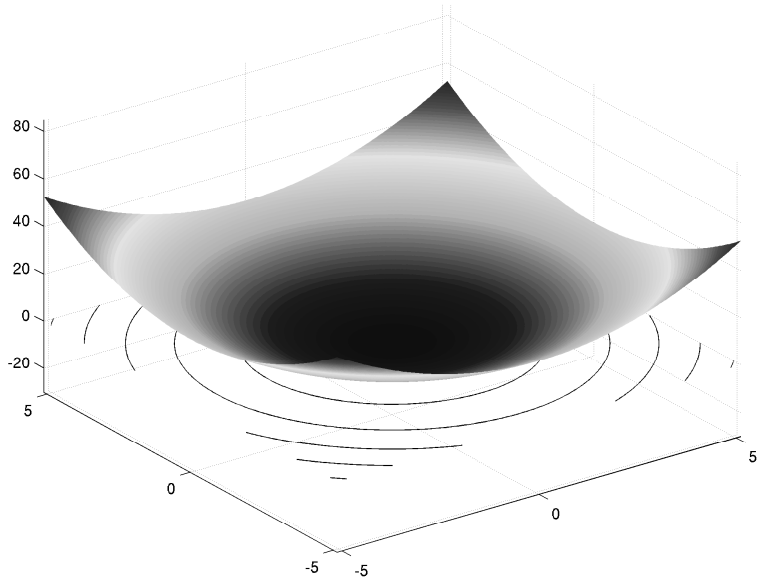


Figure 4.2: Plot of the Sphere function for *k=2* variables.

The basic, two-variable version (as given by De Jong) can be seen in Figure 4.2.

As was the case with the Rastrigin equation, this is a minimisation problem, with an optimal solution of zero.

Although De Jong's original equation only used two variables, in our experiments we used 100, thus increasing its complexity significantly. The range was set to -5.12 to 5.12 (as in De Jong's original version). As before, we used 16-bit Gray code encoding.

4.4.3 Rosenbrock Function

The Rosenbrock function is much more demanding, as its global optimum is located inside a narrow ridge, surrounded by a parabolic valley. Its general form is given in equation 4.3, and the simple, two-variable version is illustrated in Figure 4.3.

$$F(\bar{x}) = \sum_{i=1}^{k-1} [100 \times (x_i^2 - x_{i+1})^2 + (1 - x_i)^2] \tag{4.3}$$

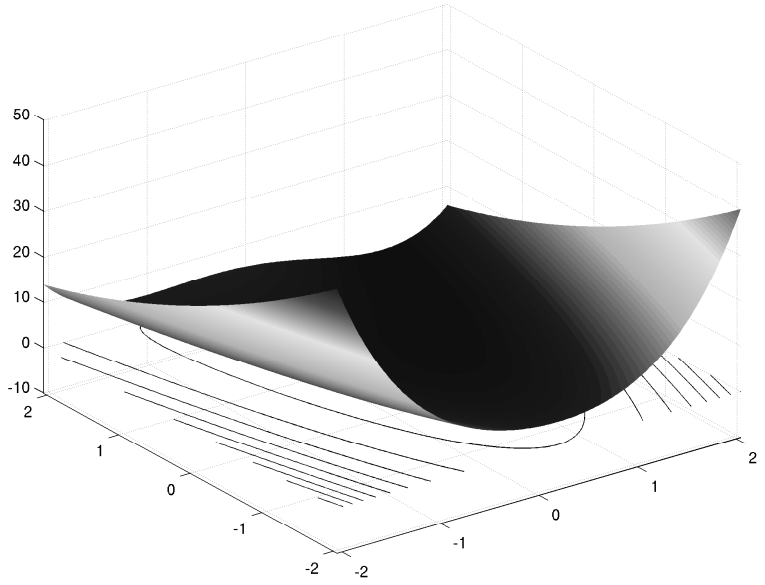


Figure 4.3: Plot of the Rosenbrock function for *k=2* variables.

This is also a minimisation problem, with an optimal solution of zero.

Again, we used 100 variables (as opposed to De Jong's 2), in order to increase the problem's complexity. The range was set to -2.048 to 2.048, and 16-bit Gray code encoding was used again.

4.5 Introducing Noise

In order to see how our system copes under noise, we introduce a number of "defective" agents in some of our configurations. These agents are defective in the following ways:

- All of their internal data (the solution genes in their intra-GA population, as well as their parameter set) remain at random values with each GA iteration. Essentially, it is impossible for a defective agent to converge, and its actual fitness remains at an arbitrary, high (i.e. bad) level.

- When asked directly about its fitness, a defective agent will provide a false, minimal (i.e. good) value. In this way, it "lures" observing peers into selecting it as a mate, even though its actual fitness is much worse than advertised.

Other than that, a defective agent behaves just as a normal one, in terms of adherence to communication protocols. As a result, a normal agent has no direct way of determining whether a peer is defective or not.

4.6 Agent Autonomy and Motivation

As is the norm in most multi-agent systems, each agent in our implementation is fully autonomous. This autonomy is evident in the fact that the agents are able to function even without any peers present, or when peer-to-peer communication is compromised. This characteristic has the obvious advantage of improved robustness.

However, an agent operating in isolation will not be able to evolve its own GA parameters, and hence its performance will remain at a steady, arbitrary level dictated by the current set of GA parameters it uses.

This is where the motivation of the agents to interact with their peers stems from: by having agents collaborate/breed with their peers, the system as a whole evolves, adapts, and improves its performance.

4.7 Summary

In this chapter, we have discussed the following points:

- The approach we took in measuring the performance of our system, in terms of the number of generations as opposed to the total number of calculations typically used in similar work.

- A brief discussion on the statistical test we performed on our results, the Mann-Whitney U-test, and how we used this to verify the validity of our findings.

- An overview of the three benchmark functions we used as optimisation problems in our experiments: A 100-variable version of the Rastrigin equation, as well as 100-variable versions of the generalised forms of two equations from De Jong's test suite: the Sphere equation and the Rosenbrock equation.
- A presentation of the behaviour of "damaged" agents, used in some experiments in order to investigate how different algorithms cope under noise.
- An explanation on why agents in our system can be considered autonomous, as well as where their motivation to interact with their peers stems from.

Chapter 5

A Peer-To-Peer Adaptive Genetic Algorithm

5.1 Overview

Having implemented a working LCC interpreter, we were able to proceed with the implementation of the first version of our algorithm - namely, a P2P parallel adaptive GA.

In this chapter, we look at the architecture of the system, as well as its configuration. We then proceed to show how our setup was used to solve a typical multi-dimensional benchmark function, the Rastrigin equation, as well as two additional benchmark functions taken from De Jong's test suite.

The results we present give us some insight into the inner workings of the system, in terms of effort distribution and parameter adaptation. In addition, we look at its performance from two different perspectives, as well as the impact of different levels of agent connectivity, and different levels of noise.

Part of the work in this chapter has been published in the proceedings of the 12th International Conference in Enterprise Information Systems (ICEIS 2010) [Chatzinikolaou, 2010].

5.2 Architecture

5.2.1 The "Intra-agent" Genetic Algorithm

The system we have developed consists of a network of an arbitrary number of identical agents. Each agent contains an implementation of a canonical GA that acts on a local population of genomes, performing standard crossover and mutation operators on them. We call this GA the "intra-agent GA", and its steps are that of a typical GA (as illustrated in Figure 5.1): For a population of size n,

1. Create a random initial population.
2. Evaluate each member of the population.
3. Select n pairs of parents using roulette wheel selection.
4. For each pair of parents, recombine them and mutate the resulting offspring.
5. Repeat from step 2 for the newly created population.

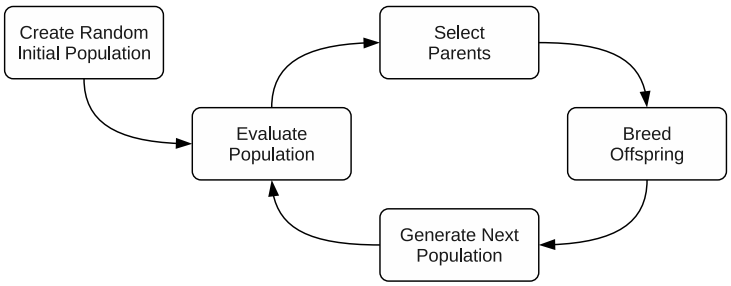


Figure 5.1: Intra-agent GA.

Since we are mainly interested in observing the adaptation that occurs in the individual agents' GAs themselves during the evolutionary process, we tried to keep things simple and controllable by only allowing a single parameter to adapt: that of the mutation rate. In this way, we were better able to observe the impact of the design of our architecture in the overall performance of the GA.

All the rest of the parameters were kept constant: the population size was fixed at 20 individuals, with an elite size of 4. The crossover operator was set to single-point crossover, with the locus selected at random along the entire length of the genome. Finally, the roulette wheel selection scheme was used exclusively for the intra-agent GA.

It must be noted at this point that the choice of these parameters was arbitrary, done with simplicity in mind rather than performance. They represent a simple, commonly-used GA configuration, and no attempt was made to fine-tune them further by scholastically exploring multiple alternatives. This was done deliberately, so as to better illustrate one of our main points - i.e. that our system requires no hand-tuned configuration prior to deployment.

5.2.2 The "Extra-agent" Genetic Algorithm

Every agent has (and executes locally) a copy of a shared, common LCC protocol that dictates how this agent coordinates and shares information with its peers. The result of this coordination is a secondary evolutionary algorithm, which evolves not the genomes in each agent but the agents themselves, and in particular the population and parameters that they use for their respective "intra-agent" GA. We call this secondary GA, illustrated in Figure 5.2, the "extra-agent" GA:

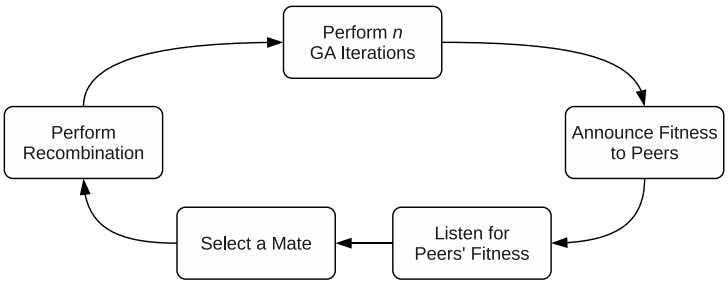


Figure 5.2: Extra-agent GA.

1. Perform a number of "intra-agent" GA iterations.

   This step is equivalent to step 2 of the "intra-agent" GA, as it essentially establishes a measure of that agent's overall fitness. This fitness is based on the average fitness of all the individual genomes in the agent's population, as established by the "intra-agent" GA.

2. Announce agent's fitness to neighbouring peers.

3. Listen for fitness announcements from neighbouring peers.

4. Select a fit mate (using either roulette wheel selection or tournament selection).

Again, this is similar to step 3 above. The only difference this time is that every agent gets to select a mate and reproduce, as opposed to the "intra-agent" GA where both (genome) parents are selected using roulette wheel selection.

5. Perform crossover and mutation operators between self and selected agent (population and parameters).

   Here we have the recombination stage between the two peers (equivalent to step 4 above), during which they exchange genomes from their respective populations (migration) as well as parameters. The new parameters are obtained by averaging those of the two peers, and adding a random mutation amount to them.

6. Repeat from step 1.

By combining the GA parameters of the agents in addition to the genomes during the migration/recombination stage (step 5), we ensure that these parameters evolve in tandem with the solution genomes, and thus remain more-or-less optimal throughout the evolutionary process.

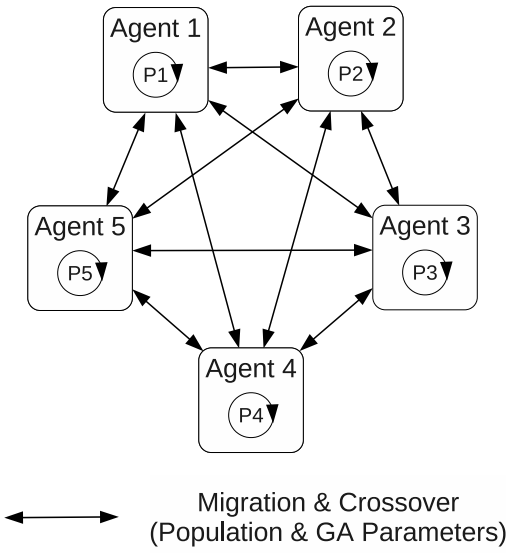


Figure 5.3: Overview of the architecture of the system.

The idea behind this approach is that we use an evolutionary algorithm to optimise the optimiser itself. Maintaining an optimal set of parameters for the optimiser, i.e. the "intra-agent" GA, can be considered as a dynamic optimisation problem in itself. By performing the standard genetic operators of selection, recombination and mutation on these parameters during the life cycle of the optimiser agents, we allow the latter to adapt to near-optimal values dynamically.

In that respect, our architecture can be viewed as a cascaded, two-level meta-GA: On the inner level (the "intra-agent" GA), we optimise the solution genomes. On the outer level (the "extra-agent" GA), we optimise the operating parameters of the inner level optimisers.

5.2.3 Agent Crossover

Three different schemes for the extra-agent crossover were used:

1. No crossover: essentially, each agent's GA ran in isolation from the others.

2. Population crossover: during the extra-agent crossover phase, only individuals between the different sub-populations were exchanged, while GA parameters were not recombined.

3. Full crossover: This is the scheme that we propose in our architecture. In this case, the parameters of the agents' GA were also recombined in addition to the population exchange.

The first two schemes were implemented not as an integral part of our architecture, but instead as a benchmark for evaluating its performance.

Essentially, scheme 1 emulates a set of traditional, canonical GAs using static parameters covering the full available spectrum. This particular configuration was tested against an additional canonical GA setup implemented using MATLAB's Genetic Algorithm toolbox (yielding similar results for similar parameters). This was done mostly as a "sanity-check", in order to ensure that our platform behaves properly. We chose MATLAB as the tool for this grounding, as it is a widely-used, independent and solid "standard" platform.

Scheme 2 on the other hand emulates a typical "island-based" GA, with migration taking place among individual GAs that - again - use static parameters.

For the first and second configuration, each agent was given a mutation rate equal to half that of the previous agent, starting at 1.0. This means that, as more agents were introduced in the system, their mutation rate was reduced exponentially. The reason for this decision is the fact that, in almost all GAs, latter generations benefit from increasingly smaller mutation rates [Eiben et al., 2000].

For the second and third case, agents were selected by their peers for crossover using either roulette wheel selection (experiments 5.3.1, 5.3.2, 5.3.3, 5.3.4 and 5.3.5)

or tournament selection with varying values for the tournament size k (experiments 5.3.6, 5.3.7 and 5.3.8), where each agent's fitness was dictated by the average fitness of its current population.

5.2.4 The Cycle Parameter

One additional parameter we had to specify was how many iterations each agent would perform before crossover with the other agents occurred (called the *cycle* parameter). This parameter is dependent on the architecture and available bandwidth available by the computational platform on which the system is deployed. Larger values represent more work done at the individual, "intra-agent" GA level, while smaller values result in more "extra-agent" breedings.

In our case, and after having tried a number of different values (ranging from 1 to 100), we set the value of *cycle* to 10, which was empirically found to allow our system to converge within manageable time frames, while at the same time allowing a sufficiently large number of "extra-agent" breedings to occur and hence help us observe the impact of the different agent selection schemes we tried for our algorithms.

5.3 Evaluation

5.3.1 Effort Distribution

Before we proceed to present the results from our actual experiments, we are going to take a look at how each extra-agent crossover scheme affects the behaviour of the agents as a system. Figures 5.4, 5.5 and 5.6 illustrate the progress of three small (100 generations) typical runs of 16 agents, using isolated, population-only and full extra-agent crossover respectively.

The no-crossover scheme (Figure 5.4) shows exactly what we expected, i.e. a series of canonical GAs with progressively smaller mutation rates, ranging from "fast-and-rough" (upper agents, bigger mutation rate) to "slow-and-precise" (lower agents, smaller mutation rate).

The "stepping" effect observed for the population-only scheme (Figure 5.5), more pronounced for the lower agents (smaller mutation rate), is a typical characteristic of this scheme, with steps occurring after every extra-agent crossover cycle (10 generations).

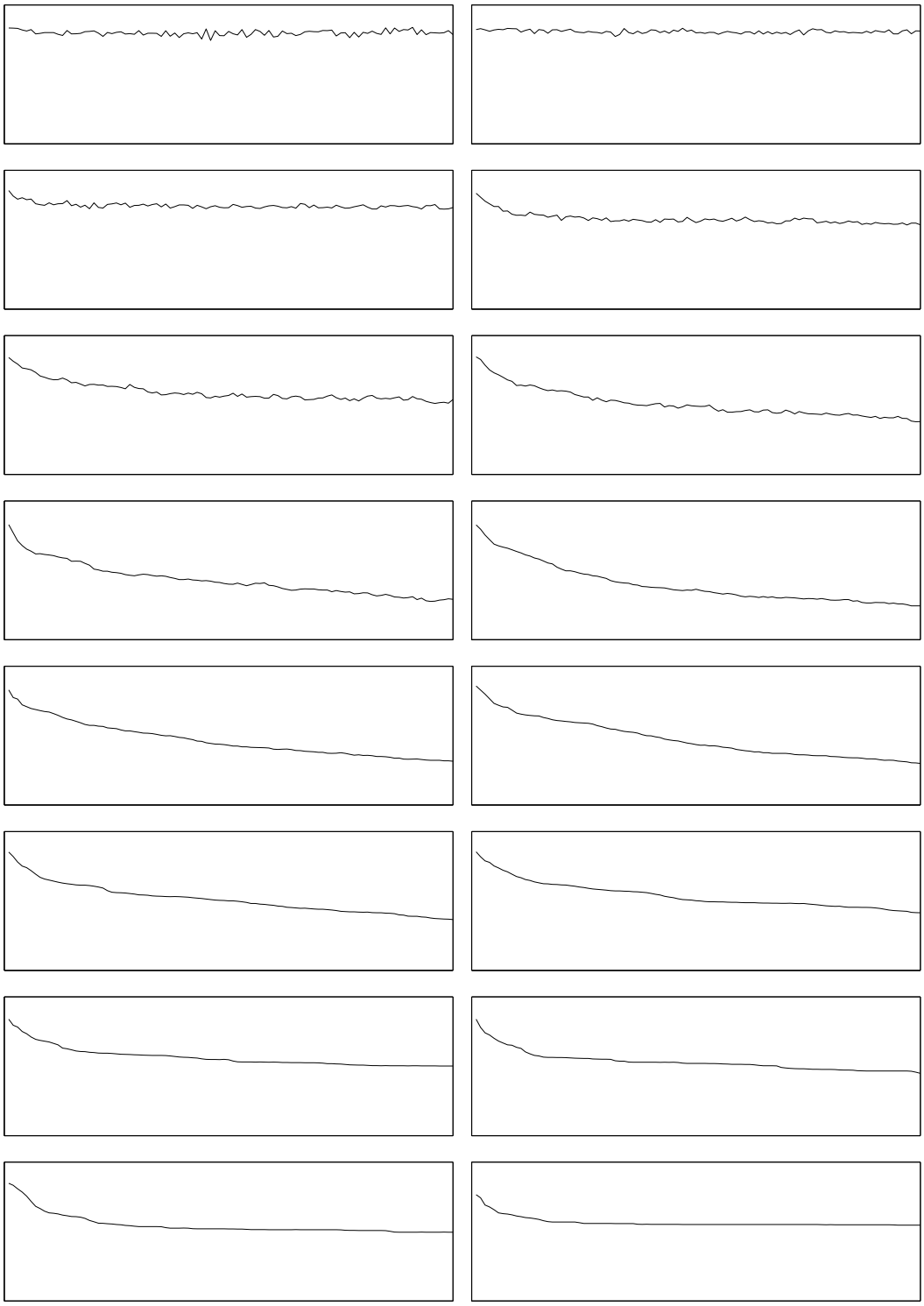


Figure 5.4: Small (100 generations) test run with 16 agents and no extra-agent crossover. X-axis is generation, Y-axis is average fitness.

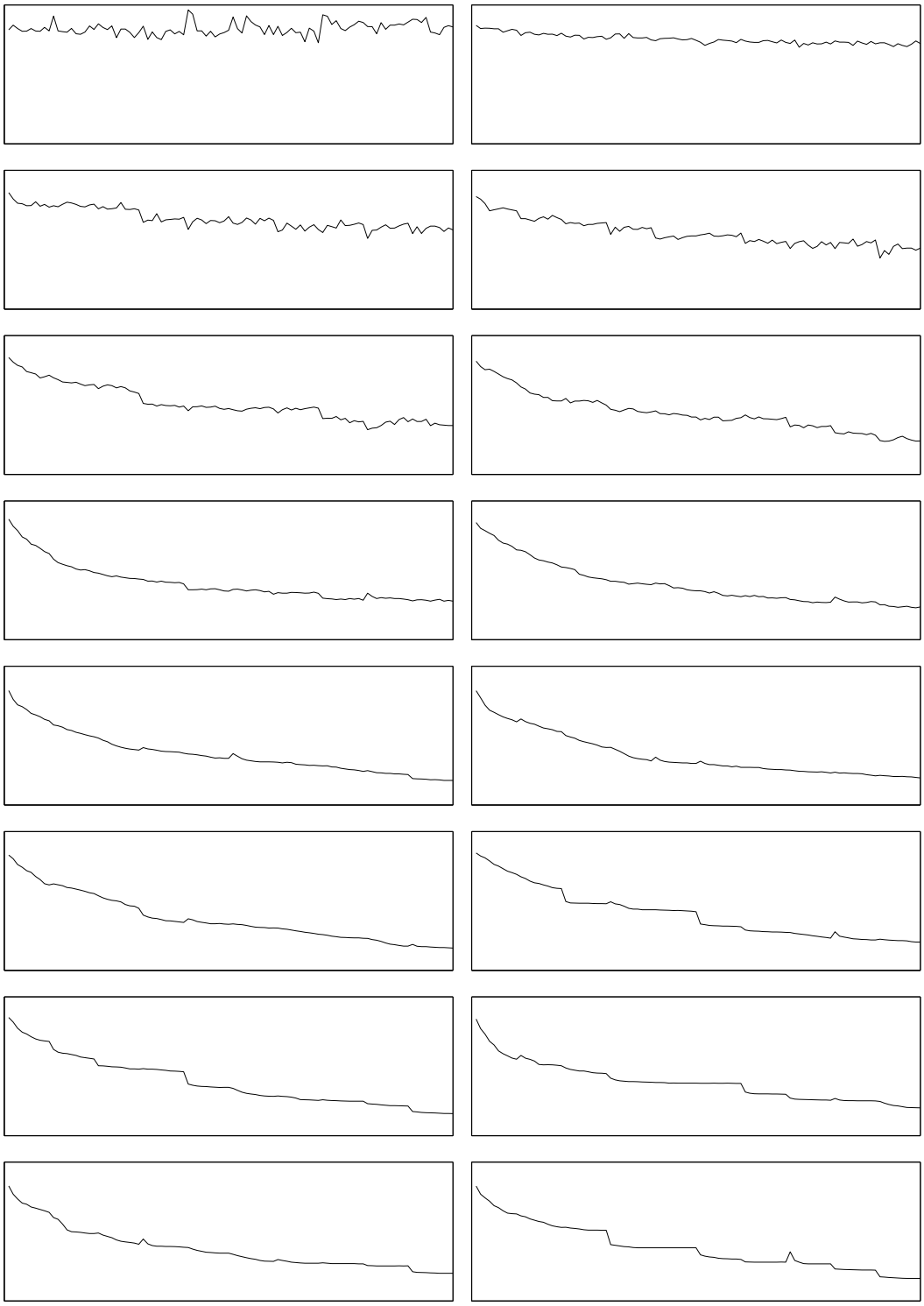


Figure 5.5: Small (100 generations) test run with 16 agents and population-only extra-agent crossover. X-axis is generation, Y-axis is average fitness.

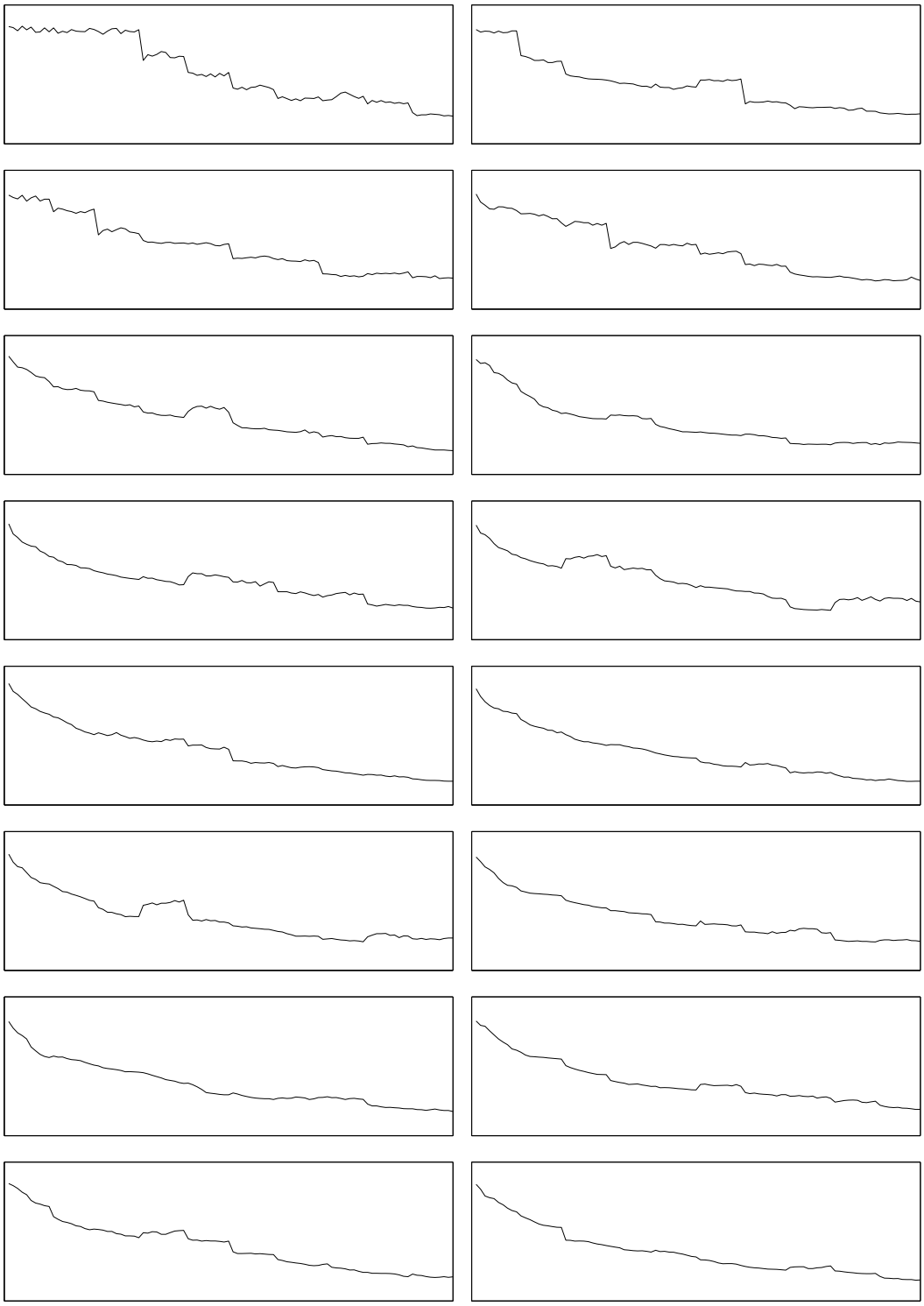


Figure 5.6: Small (100 generations) test run with 16 agents and full extra-agent crossover. X-axis is generation, Y-axis is average fitness.

Finally, the adaptation of the mutation rate is evident for the full-crossover scheme (Figure 5.6), with upper agents (with initially high mutation rates) progressively becoming equally effective as their peers.

5.3.2 Parameter Adaptation

As a further investigation on the behaviour of the system, and in particular on how the mutation rate adapts in the full extra-agent crossover scheme, we plotted the progress of the best agent's mutation rate against the generations, in a typical run using eight agents.

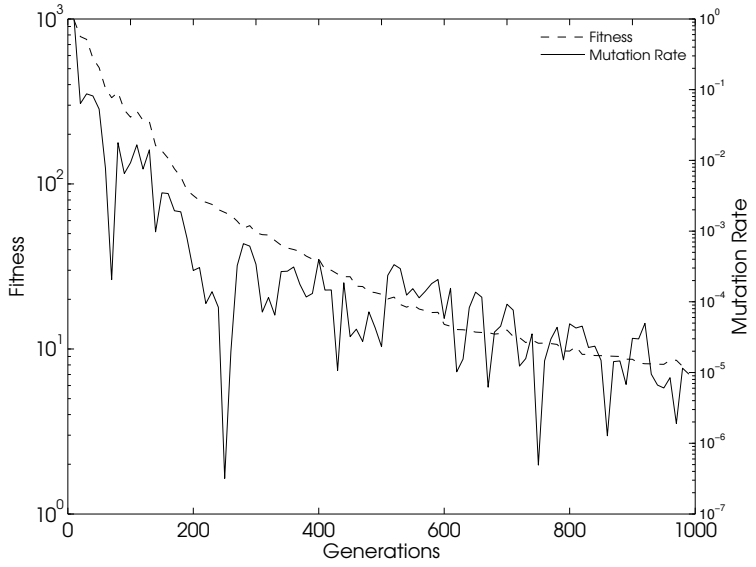


Figure 5.7: Adaptation of the mutation rate (one of eight agents).

Figure 5.7 illustrates the results (again using a logarithmic y-axis). From this plot, we can see that the mutation rate drops more-or-less exponentially in order to keep minimising the fitness, which agrees with our expectations.

5.3.3 Quality of Solution

Our first performance-oriented experiment involved executing runs for 1000 generations each, again using all three extra-agent crossover schemes for different numbers of agents. This allowed us to see how close to the optimal fitness of 0.0 each configuration converged.

The graphs in Figure 5.8 shows the resulting graphs from these runs, with the actual fitness results provided in Table 5.1. The y-axis of the graphs has been made

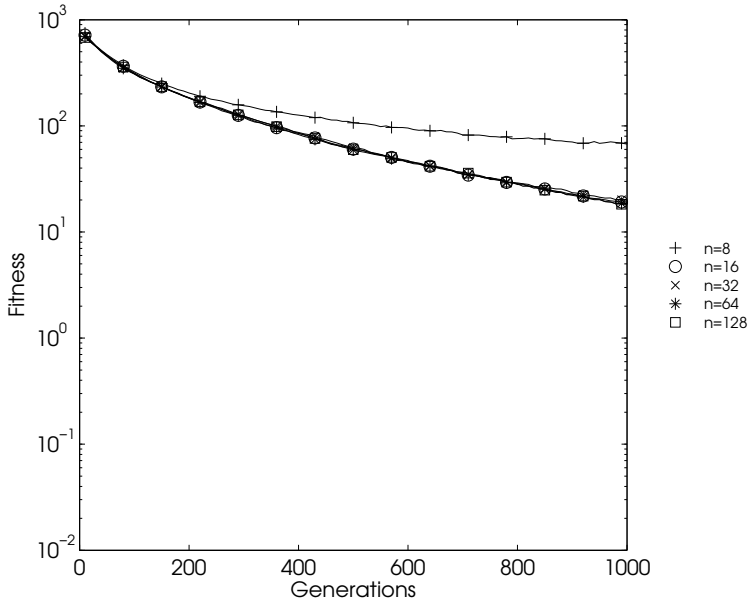
logarithmic in order to improve the legibility of the plots.

Regarding the no-crossover scheme, it can be seen that, for $n > 16$, its performance remains identical ($0.4 < p < 0.8$). By this we can deduce that the optimal (fixed) mutation rate appears within the first 16 agents, so adding more makes no difference.

Again for $n > 16$, the population crossover scheme performs significantly better than the no-crossover scheme ($p < 0.01$ in all cases), although adding more agents seems to make little difference, as the performance for different numbers of agents remains similar. In fact, $n = 128$ performs consistently worse than $n = 64$ ($p < 0.01$), which performs consistently worse than $n = 32$ ($p < 0.01$).

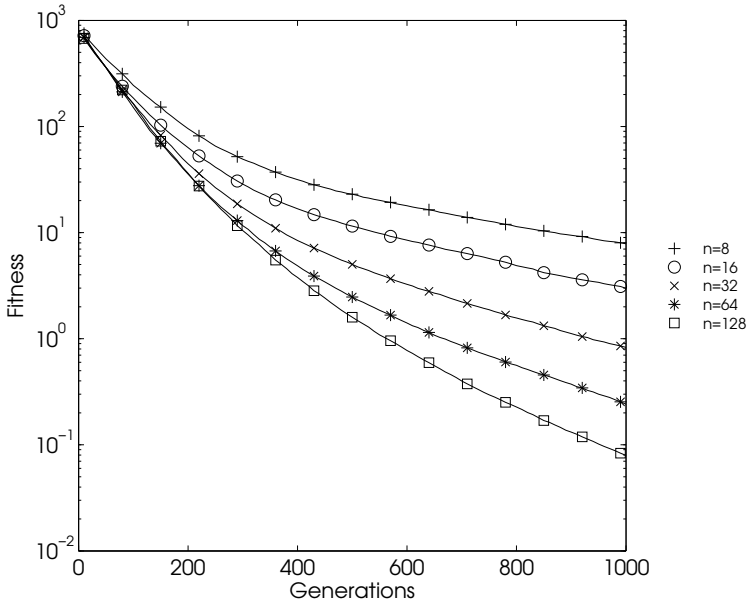
Crossover	Best Fitness
None	17.94 (at $n = 128$)
Population	1.84 (at $n = 32$)
Full	0.08 (at $n = 128$)

Table 5.1: Best (minimum) fitness after 1000 generations.

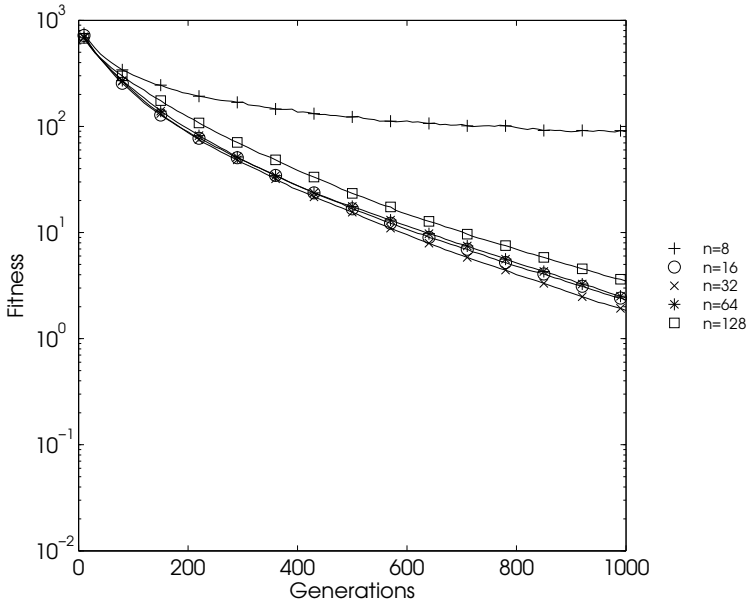


(a) Run of 1st scheme (no extra-agent crossover) for different numbers of agents.

Figure 5.8: Best (minimum) fitness after 1000 generations.



(b) Run of 3d scheme (full extra-agent crossover) for different numbers of agents.



(c) Run of 2nd scheme (population extra-agent crossover) for different numbers of agents.

Figure 5.8: Best (minimum) fitness after 1000 generations.

In contrast, the full crossover scheme scales significantly better as the number of agents increases. The first two schemes seem to be "hitting a wall" after the number of agents is increased beyond 16. For the case of the full crossover, however, adding

more agents results in a significant increase in the performance of the system all the way up to, and including, $n = 128$ ($p < 0.01$).

By comparing all three graphs, it becomes obvious that using the full crossover scheme achieves the best solution in terms of quality, in addition to being the fastest of the three.

Finally, the ability of this scheme to perform well even when using a small number of agents can also be seen graphically in Figure 5.8b.

5.3.4 Speed of Convergence

For our next performance-oriented experiment, we executed runs using different numbers of agents and all three extra-agent crossover schemes. Each run was stopped as soon as a fitness of 1.0 (or the limit/max generation of 10,000) was reached by any of the agents in that run.

The no-crossover scheme, representing an isolated, canonical GA, reached the target after 1991 generations (with a standard deviation of 159) when run for a sufficiently large number of agents ($n > 16$), which ensures that at least one GA instance having an optimal mutation rate is included.

The results for the other two crossover schemes are given in Table 5.2, while Figure 5.9 illustrates graphically the relative performance of the three schemes (note that the x-axis is shown in logarithmic scale).

Crossover	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
Population		1179 *(121)*	1136 *(80)*	1246 *(70)*	1379 *(91)*
Full	2242 *(431)*	1451 *(291)*	920 *(111)*	665 *(56)*	549 *(34)*

Table 5.2: Relative speed performance of the two extra-agent crossover schemes. Averaged values (σ in parentheses).

As can be seen, the slowest performer was the first scheme, which emulates a number of isolated sequential GAs.

The population exchange scheme performed significantly better in terms of speed for all numbers of agents ($p < 0.01$ in all cases). However, it failed to converge when

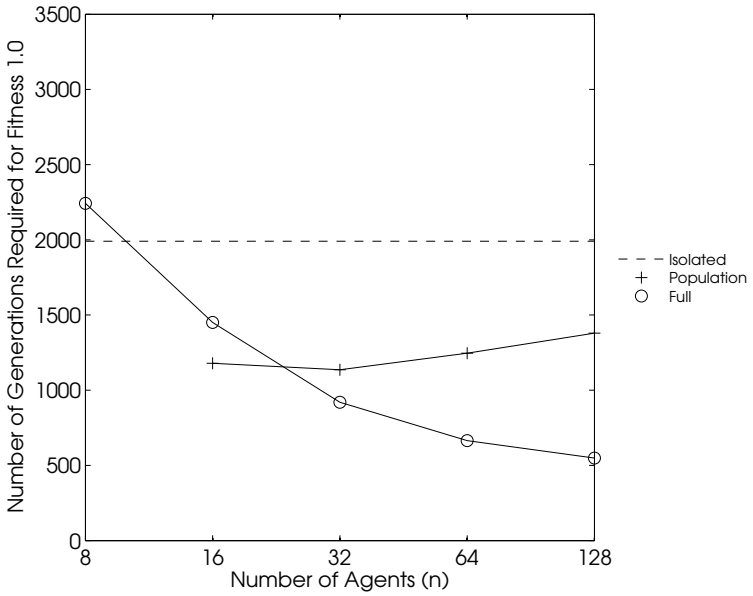


Figure 5.9: Relative speed performance of the two extra-agent crossover schemes.

few ($n = 8$) agents were used - the reason for this being that the agents' (fixed) mutation rates were too high to allow them to converge to the target fitness.

The full crossover scheme performed even better in terms of speed (except for $n = 16$), but its most significant advantage is the fact that it managed to reach the target fitness even when using few agents - although at the expected cost of more generations.

Finally, the downward slope of this scheme's curve as the number of agents increases, provides another hint of its improved scaling properties.

5.3.5 Additional Benchmarks

In order to ensure that our system behaves consistently across a range of different optimisation problems, we performed experiments using the two additional benchmark functions presented in 4.4.

Table 5.3 and Figure 5.10 show the results for the Sphere function, using population and full crossover. The isolated, no-crossover scheme reached the target fitness of 1.0 in 1881 generations (with a standard deviation of 187).

As can be seen, the relative performance of the three crossover schemes remain identical to the ones for the Rastrigin function, presented in the previous section.

Table 5.4 and Figure 5.11 show the corresponding results for the Rosenbrock function. This time, the target fitness was set to 350, since this benchmark function tended to produce fitness values several orders of magnitude higher than the previous two (for

Crossover	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
Population		1087 *(138)*	1062 *(61)*	1117 *(73)*	1258 *(93)*
Full	2128 *(467)*	1366 *(364)*	805 *(101)*	581 *(54)*	526 *(25)*

Table 5.3: Sphere benchmark function: Relative speed performance of the two extra-agent crossover schemes. Averaged values (σ in parentheses).

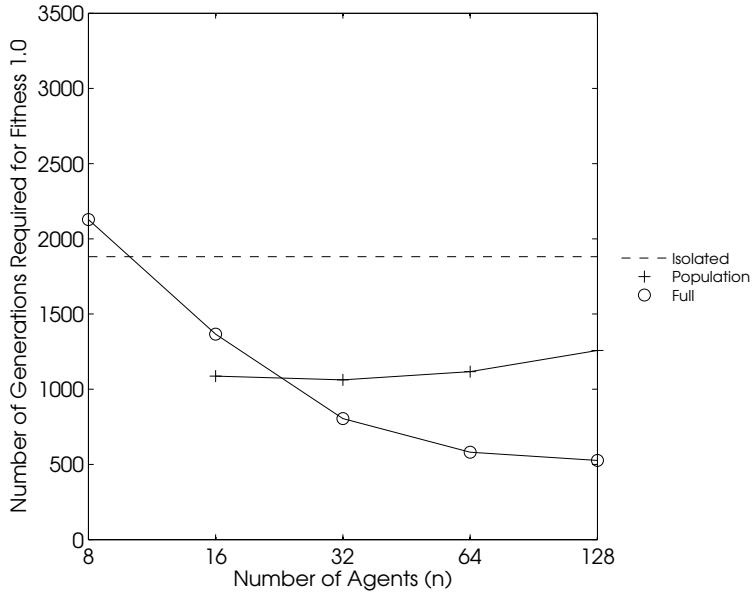


Figure 5.10: Sphere benchmark function: Relative speed performance of the two extra-agent crossover schemes.

a roughly equal number of generations), in earlier as well as in latter generations. The isolated GA (no-crossover scheme) reached this target after 1916 generations (with a standard deviation of 334).

It is interesting to note here that the standard deviations for all cases are higher than in the Rastrigin and Sphere functions. This shows how successful this function is at trapping a GA into local minima.

Despite this, the relative shapes of the curves remain unchanged, which shows that the system performs consistently across different optimisation problems.

Crossover	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
Population		1182 *(431)*	1137 *(572)*	991 *(411)*	984 *(241)*
Full	2326 *(1223)*	1195 *(685)*	539 *(188)*	356 *(56)*	329 *(25)*

Table 5.4: Rosenbrock benchmark function: Relative speed performance of the two extra-agent crossover schemes. Averaged values (σ in parentheses).

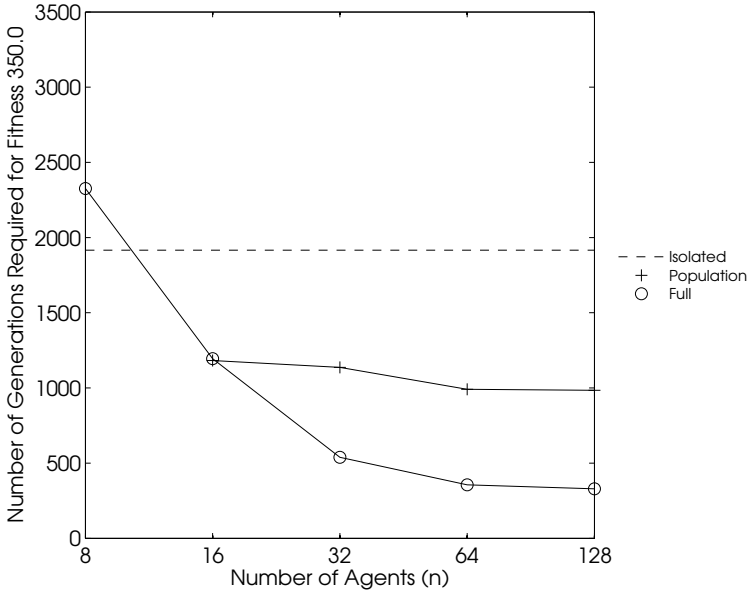


Figure 5.11: Rosenbrock benchmark function: Relative speed performance of the two extra-agent crossover schemes.

5.3.6 Connectivity

In all previous experiments, agents used roulette wheel selection in order to select a mate from among their peers. This assumes a fully-connected network of peers, which, in large-scale applications, is not practical.

For our next experiment, we substituted roulette wheel selection with tournament selection, for varying values of k (tournament size). This allowed us to determine how varying levels of connectivity between peers affect the performance of the system.

k	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
1	2027 *(703)*	1731 *(746)*	1565 *(662)*	1421 *(734)*	1032 *(563)*
2	2249 *(649)*	1481 *(311)*	898 *(151)*	675 *(69)*	532 *(51)*
25%	2249 *(649)*	1437 *(247)*	1037 *(144)*	707 *(65)*	566 *(33)*
50%	2217 *(455)*	1527 *(288)*	963 *(135)*	687 *(71)*	538 *(32)*
100%	2363 *(517)*	1426 *(250)*	941 *(159)*	656 *(69)*	564 *(29)*

Table 5.5: Full crossover scheme (no defective agents): Connectivity results. Averaged values (σ in parentheses).

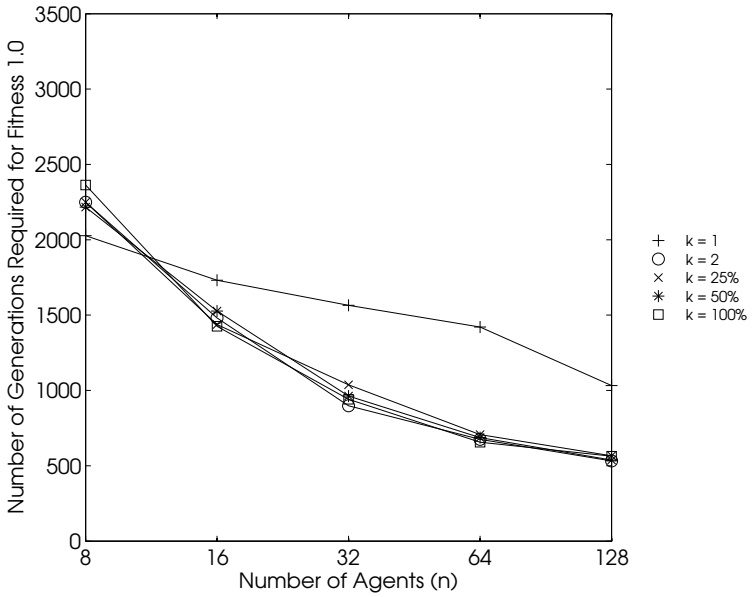


Figure 5.12: Full crossover scheme (no defective agents): Connectivity results.

The results from this experiment can be seen in Table 5.5 and Figure 5.12.

From these results, we can see that the level of connectivity, at least in the absence of defective agents, does not affect the performance of the system significantly. The only exception to this is the $k = 1$ case, which essentially constitutes random selection.

As expected, its performance was inferior for all values of n, except for few agents ($n = 8$) where it performed better than larger values of k ($p < 0.01 < 0.15$).

That case aside, the largest discrepancy occurs at $n = 32$, where asking 2 peers is significantly better than asking 25% or 50% peers ($p < 0.05$), although less significantly better than asking all peers ($p > 0.25$).

5.3.7 Connectivity Under Noise

The results in the previous section change dramatically when we introduce noise in the system, in the form of a single defective agent. Consider the results presented in Table 5.6 and Figure 5.13:

k	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
2		2688 *(1154)*	921 *(129)*	579 *(36)*	504 *(36)*

Table 5.6: Full crossover scheme (one defective agent): Connectivity results. Averaged values (σ in parentheses).

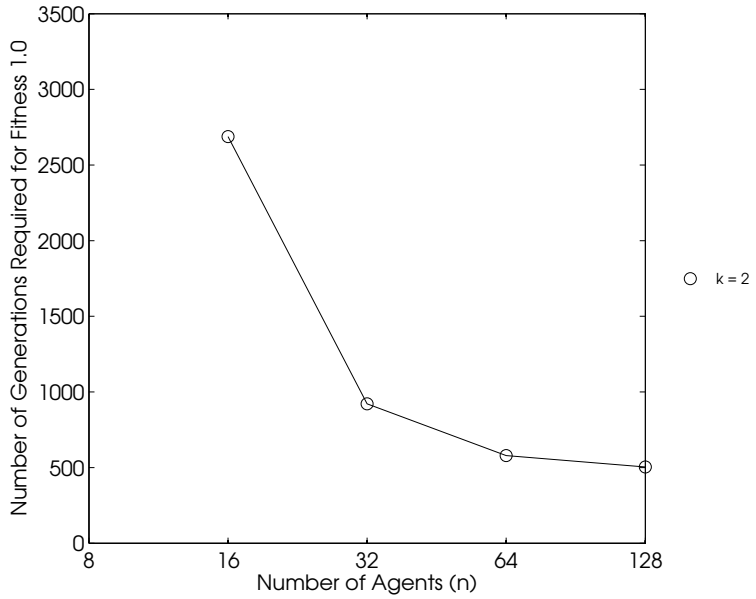


Figure 5.13: Full crossover scheme (one defective agent): Connectivity results.

The most striking difference is that the only configuration that consistently managed to converge to the target fitness is $k = 2$, i.e. binary tournament selection. This finding agrees with [Miller et al., 1995].

Higher values of k increased the possibility that the damaged agent would end up in the tournament, which would invariably cause it to get selected as a mate (since it falsely advertises a very good fitness) - resulting in increasingly worse performance and a higher proportion of failed runs. Random selection ($k = 1$), on the other hand, also failed to converge for any agent population size (it almost managed for $n = 128$, but a small proportion of runs failed).

Comparing the graphs in Figures 5.12 and 5.13 for $k = 2$ gives us another interesting result: For a small population of agents ($n = 8$), no value for k succeeded in reaching the target fitness under noise. As the agent population size is increased, however, we notice an interesting trend: For $n = 16$, adding noise significantly decreased the performance of the system ($p < 0.01$). For $n = 32$, the performance remained virtually unaffected ($p > 0.35$). For larger agent population sizes ($n \geq 64$), adding a damaged agent caused the system to perform significantly better ($p < 0.01$).

However, this finding is not inconsistent with evolutionary theory, which maintains that a small amount of noise can in fact be beneficial in evolution, as it increases variation. This is the main principle behind the use of the mutation operator in GAs.

5.3.8 Noise Profile

In order to obtain a more complete noise profile for the fitness-based reputation, we performed a series of runs for varying agent population sizes n and levels of noise d (up to a maximum of 50%), where each configuration was allowed to run up to the cut-off generation of 10,000. In all cases, we used the full crossover scheme, with a tournament size $k = 2$.

The results are given in Table 5.7, and can be seen graphically in Figure 5.14.

By observing the graph in Figure 5.14 it becomes apparent that, although the system can cope with (and in fact benefits from, as seen in section 5.3.7) a small percentage of noise (up to around 3% for $n \geq 32$), raising the level of noise further by adding more damaged agents causes it to break down rather quickly.

For very high levels of noise ($d = 50\%$), the system fails to optimise the problem to any significant degree, i.e. the fitness remains more-or-less at the initial levels. This limitation is not alleviated even for larger agent population sizes.

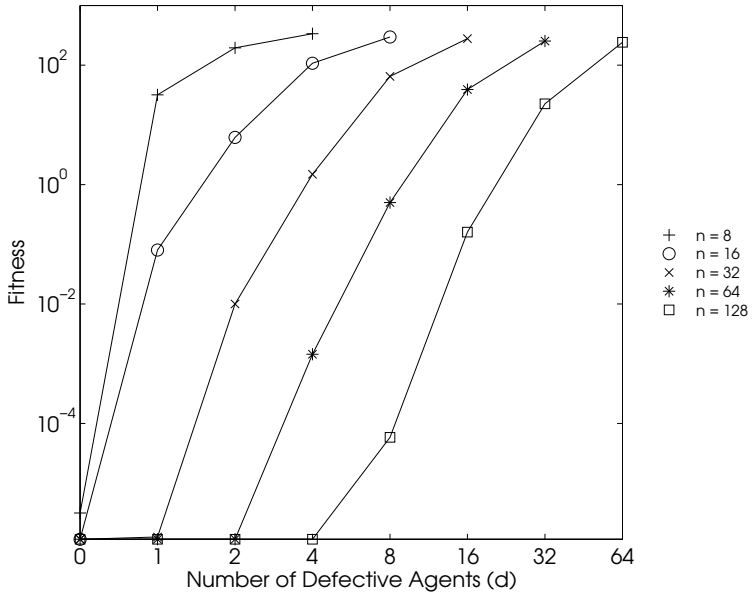


Figure 5.14: Noise profile for the fitness-based algorithm ($k = 2$).

d	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
0	3.16E-6 *(9.23E-6)*	1.15E-6 *(0)*	1.15E-6 *(0)*	1.15E-6 *(0)*	1.15E-6 *(0)*
1	31.83 *(26.07)*	0.08 *(0.12)*	1.25E-6 *(3.11E-7)*	1.15E-6 *(0)*	1.15E-6 *(0)*
2	195.04 *(51.87)*	6.17 *(5.17)*	0.01 *(0.02)*	1.15E-6 *(0)*	1.15E-6 *(0)*
4	337.05 *(65.84)*	107.92 *(43.16)*	1.49 *(1.53)*	1.44E-3 *(2.67E-3)*	1.15E-6 *(0)*
8		296.23 *(54.08)*	64.85 *(28.35)*	0.50 *(0.38)*	5.85E-5 *(9.46E-5)*
16			278.86 *(36.27)*	39.18 *(11.24)*	0.16 *(0.10)*
32				253.76 *(36.22)*	22.53 *(6.48)*
64					241.38 *(23.25)*

Table 5.7: Noise profile for the fitness-based algorithm ($k = 2$).

5.4 Discussion

The results presented above are encouraging, as they demonstrate that this preliminary version of the architecture we propose is effective. By distributing the load among multiple agents, the system manages to converge to near-optimal solutions in relatively few generations compared to a canonical GA. In addition, the peer-to-peer architecture of the system provides inherent benefits such as improved robustness and scalability.

By applying the principle of natural selection to optimise the GA agents themselves, the evolutionary algorithm becomes adaptive, thus eliminating the need for hand-tuning (although in our investigation this was restricted to the mutation rate).

Finally, when using binary tournament selection rather than a higher bias scheme like roulette wheel selection, the system was able to cope with, and in some cases even benefit from, low levels of noise, which was introduced in the form of defective agents.

5.5 Summary

In this chapter, we have discussed the following:

- A detailed description of our parallel GA architecture; in particular, the inner level "intra-agent" GA and the outer level "extra-agent" GA.
- The fundamental differences between our system and other existing evolutionary algorithm approaches.
- The experimentation setup, including parameter setup and a description of the three crossover schemes that we tested.
- The results of our experiments, showing:

  Observations into how the system distributes the effort of the evolutionary process among agents using the three extra-agent crossover schemes we tested.

  Observations into how the mutation rate adapts during runtime, illustrating the adaptive properties of our algorithm.

  How our system performs in terms of quality of solution (achieved after a fixed number of generations) and speed of convergence (towards a fixed target fitness).

  How the performance of our system remains consistent for different benchmark functions.

The impact that different levels of agent connectivity have on performance, when using tournament agent selection, in both noise-free and noisy configurations.

A comprehensive noise profile, illustrating how different agent population sizes perform for increasing levels of noise.

Chapter 6

Reputation as a Fitness Indicator

6.1 Overview

Following the implementation of the adaptive P2P GA presented in the previous chapter, we were able to proceed to the next step of our investigation.

In this chapter, we show how we modified our algorithm to use a number of non-heuristic, probabilistic reputation models as the selection bias for evolution, instead of the traditional, direct fitness reporting/observation approach which is common in most evolutionary algorithms. We describe the architecture of the reputation models as well as their configuration, and then proceed to present the results of the various experiments that we conducted in order to test the efficacy of this new approach.

These results allow us to directly compare the different versions of our algorithm in terms of performance, and investigate the effect of different levels of connectivity between the cooperating agents. In addition, we demonstrate the way and extent in which our reputation-based approach manages to deal with noise, added to the system in the form of "defective" agents.

Part of the work in this chapter has been published in the proceedings of the 12th Genetic and Evolutionary Computation Conference (GECCO 2012) [Chatzinikolaou and Robertson, 2012].

6.2 Architecture

6.2.1 Adding Trust and Reputation

It has been shown [Schillo et al., 2000, Sakai et al., 2005, Du and Fu, 2011] that in an open, peer-to-peer multi-agent environment, where individual agents cannot be controlled or guaranteed to be what they should be, reputation can be used as a mechanism to weed out defective or malicious agents.

We test this by implementing and testing three different trust / reputation models as alternatives to the traditional direct fitness observation selection mechanism. These models differ in scope (i.e., where the reputation information is stored), as well as performance and level of resistance to noise (defective/malicious agents).

In order to implement these models, the only additional requirements from the part of the agents are:

- For all three models: The ability of agents to identify their peers (each peer is allocated a unique, guaranteed-to-be-true identifier).
- For the "memory" and "collective" models: A simple form of associative memory (in our case, a hash map).

6.2.2 The Reputation Models

Memory

The first reputation model we tested was the simplest, and also the one with the most limited scope - the reputation information acquired by each agent is stored within that agent alone, and is not shared with its peers. For this reason, strictly speaking this is a *trust* model rather than a reputation model, as described in section 2.4.2.

The "memory" algorithm, illustrated schematically in Figure 6.1, works in the following way:

1. Perform a number of "intra-agent" GA cycles.
2. Locally record the (cumulative) gain/loss in fitness that resulted from the last interaction (mating) with the previously selected mate.
3. Select a new mate from the recorded history of past experiences (using tournament selection).

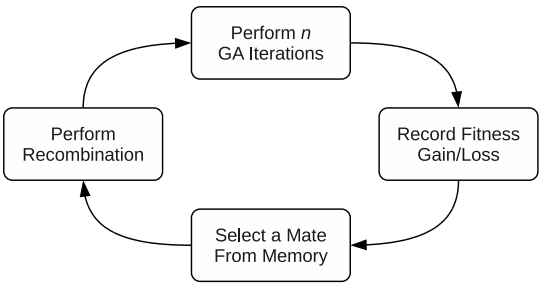


Figure 6.1: Memory Reputation GA.

4. Perform recombination of population AND parameters with the selected peer.

The "memory" model has the smallest requirements in terms of bandwidth used for selection: no transactions are required for a mate to be selected.

Central

This model is a true reputation mechanism, as agents now rely not only on their own experience, but also on their peers. This is achieved by having a centralised database where each agent reports the results of their interactions with their peers, and using this in order to select future mates.

This model differs from the "memory" model in the following:

- Shared reputation information is more complete and reliable than individual trust information, assuming that all agents have the same motivation (which, in our case, they do - defective agents notwithstanding).
- The existence of a centralised reputation database is problematic in a peer-to-peer system, as it requires distribution of that database, in addition to increasing the points-of-failure for the system.
- More communication bandwidth is required, as agents need to contact the centralised database in addition to their peers during breeding.

The "central" algorithm, illustrated schematically in Figure 6.2, works in the following way:

1. Perform a number of "intra-agent" GA cycles.
2. Report the gain/loss in fitness that resulted from the last interaction (mating) with the previously selected mate to a (cumulative) central, shared database.

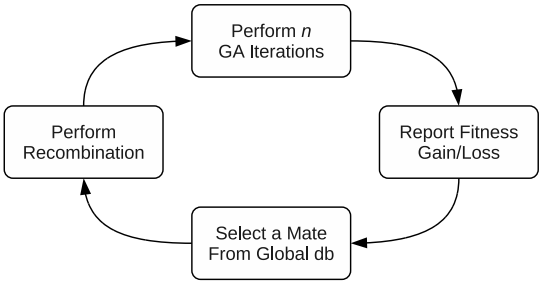


Figure 6.2: Central Reputation GA.

3. Select a mate from the same central database (using tournament selection).

4. Perform recombination of population AND parameters with the selected peer.

In terms of the bandwidth required by each peer to select a mate, the "central" model is more expensive than the "memory" model, requiring two transactions (with the central database) for each mating: One for requesting advice from the database, and one for updating it.

Collective

The third model we tried is similar in scope with the "central" model, as here agents also depend on the experience of their peers, in addition to their own, in order to select a fit mate. The significant difference is that this is a purely decentralised model, without a centralised database in which these experiences are stored. Instead, agents ask their peers directly for their "suggestions". The implications of this are:

- Defective/malicious agents have no way to sabotage the central database by reporting false results to it. This increases the robustness compared to the "central" model.

- The system requires more communication bandwidth than the other two models, especially when agents aggregate reputation information from many of their peers.

- The reputation information that agents obtain - especially in sparsely connected networks - is less complete, since it takes more iterations for each agent to "test" every one of its peers than it would take for more peers reporting in parallel to a central database. The end result of this is that the reputation information should be more out of date than in the "central" model.

The "collective" algorithm, illustrated schematically in Figure 6.3, works in the following way:

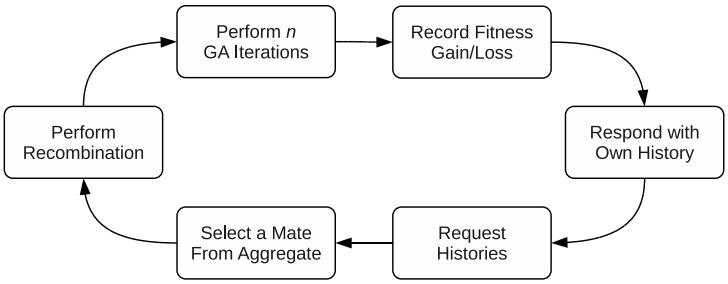


Figure 6.3: Collective Reputation GA.

1. Perform a number of "intra-agent" GA cycles.
2. Locally record in own history the gain/loss in fitness that resulted from the last interaction (mating) with the previously selected mate.
3. Respond with own recorded history to inquiring peers.
4. Select one or more (according to the value for k used) peers from the best ones in own history, and ask each one for their recorded history.
5. Aggregate own history with those peers' history, and select a mate from there (using tournament selection).
6. Perform recombination of population AND parameters with the selected peer.

The "central" model is potentially the most expensive one in terms of selection bandwidth, requiring anywhere from one to $n-1$ transactions per peer per selection, depending on the value of k used.

6.2.3 Reputation Selection Pressure

An interesting problem that we came up with while implementing these reputation models was deciding on which selection scheme to use whenever an agent has to select a mate from among its peers, whether it is from local, global or collective reputation data.

Whenever we tried a low selection pressure scheme such as binary tournament selection, performance was good for low levels of damage, but quickly degraded as

we introduced more defective agents - since these ended up getting selected relatively often. Using a high selection pressure scheme such as roulette wheel selection (or a large size tournament selection), defective agents were avoided more successfully, but for low levels of noise agents kept stuck with selecting only a few of their peers for mating, which meant that a high percentage of peers were left isolated - with a significant negative impact on performance, since this meant a reduction in variation as well as effort distribution.

Inspired by the findings presented in [Miller et al., 1995], the solution we came up with was to adjust the selection pressure according to the level of noise in the system. We achieved this by using tournament selection with a tournament size s which is not fixed, but instead it is a function of the number of damaged agents d in a population of size n (limited in $[2,n)$), as shown in Equation 6.1.

$$s = \begin{cases} 2, & \text{if } d < \frac{2}{\alpha} \\ [\alpha \times d], & \text{if } \frac{2}{\alpha} \leq d \leq \frac{n}{\alpha} - 1 \\ n-1, & \text{if } d > \frac{n}{\alpha} - 1 \end{cases} \qquad \text{for } d \in [0,n] \tag{6.1}$$

The logic behind this, from an evolutionary perspective, is that a low selection bias (small s) is good for fast evolution (more variation), whereas a high selection bias (large s) is better at leaving out bad individuals.

The system remains entirely stochastic, but this parametrised selection scheme does assume that we know a priori how many damaged agents there are in the network. However, this does not seem to be an unreasonable assumption to make, as even in a real-world system we would probably have a rough idea about the percentage of damaged nodes that are likely to be present.

We experimented with various values for α, between 0.5 and 2 (since, in our experiments, we go up to 50% damaged agents). The differences were not very pronounced, so we decided on $\alpha = 2$ which gave slightly better noise tolerance behaviour.

6.3 Evaluation

6.3.1 Coping with Noise

Observing the system in real-time gave us some insight on how a reputation-based model (in this instance, the "collective" one) manages to cope with the defective agent:

In the first few generations, the agents have not yet acquired, collectively or individually, enough experiential information that would allow them to favour the better mates. Further, they are still oblivious to the presence of the defective peer. As a result, every agent, including the defective one, has the same chances of getting selected as a mate.

As the evolution progresses, however, and as the agents "realize" which agents are better (at least, at that particular instance in time), the defective one tends to get selected increasingly less often, up to the point that it gets selected rarely enough to not be able to cause widespread damage by breeding.

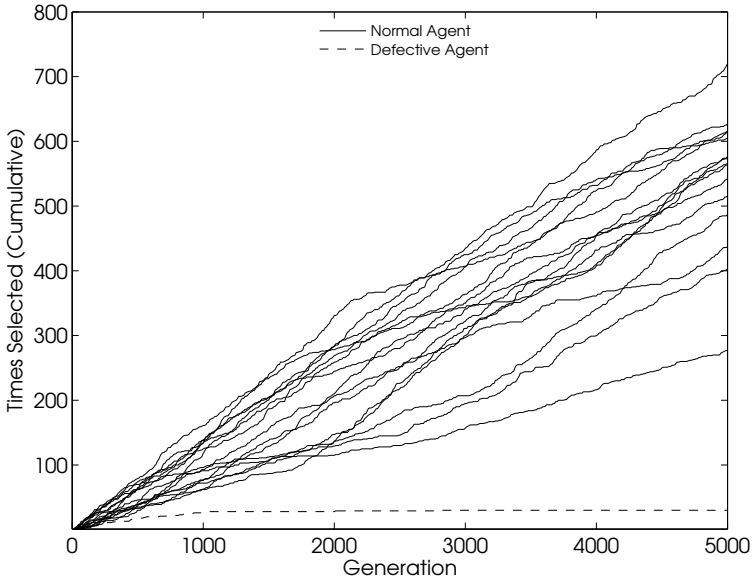


Figure 6.4: Selection frequency graph for a 16-agent run with one defective agent.

Occasionally, the defective agent will get chosen again in the latter stages in the evolution, and spread its damage to many or even all of its peers. However, invariably this phenomenon will eventually become rare enough so as to be insignificant.

This behaviour can be seen graphically in Figure 6.4, which shows results from a typical 16-agent run with one defective agent. On the X-axis is the generation time line, while the values on the Y-axis denote the cumulative number of times each agent got selected by its peers for breeding.

6.3.2 Speed of Convergence

In this next experiment we compare the performance of the three reputation models against the fitness-based model discussed in Chapter 5 (using full extra-agent crossover in all cases). Both the fitness-based model and the "collective" reputation-based model are fully connected at this stage ($k = n - 1$). As a further baseline benchmark, we included the results from the isolated version in the graph, which reached the target fitness after 1991 generations ($\sigma = 159$)

In all cases, we performed runs using 8, 16, 32, 64 and 128 agents. As was the case with the results presented in 5.3.4, each run was stopped as soon as the target fitness of 1.0, or the max/limit of 10,000 generations, was reached.

The results from this comparison are given in Table 6.1. Figure 6.5 contrasts the performance of all four models graphically.

Model	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
Fitness	2242 *(431)*	1451 *(291)*	920 *(111)*	665 *(56)*	549 *(34)*
Memory	1732 *(490)*	1596 *(663)*	1427 *(637)*	1427 *(1044)*	854 *(508)*
Central	1881 *(509)*	1346 *(445)*	1130 *(398)*	916 *(364)*	605 *(135)*
Collective	2026 *(509)*	1163 *(306)*	912 *(310)*	719 *(204)*	524 *(87)*

Table 6.1: Speed performance of the three reputation-based models versus the fitness-based model. Averaged values (σ in parentheses).

The "memory" model proved to be the worst performer, yielding inferior results to all other algorithms for $n > 16$ ($p < 0.05$). At $n = 16$ it performed roughly the same with the fitness-based model ($p > 0.8$), but significantly worse than the other two reputation models ($0.01 < p < 0.15$). However, for $n = 8$, it was the best performer ($0.01 < p < 0.2$).

For $n > 8$, the "central" reputation model performed better than "memory" ($0.01 < p < 0.15$). Compared to the fitness-based model, however, it performed worse for $8 < n < 128$ ($0.01 < p < 0.15$), about the same for $n = 128$ ($p > 0.7$), and significantly

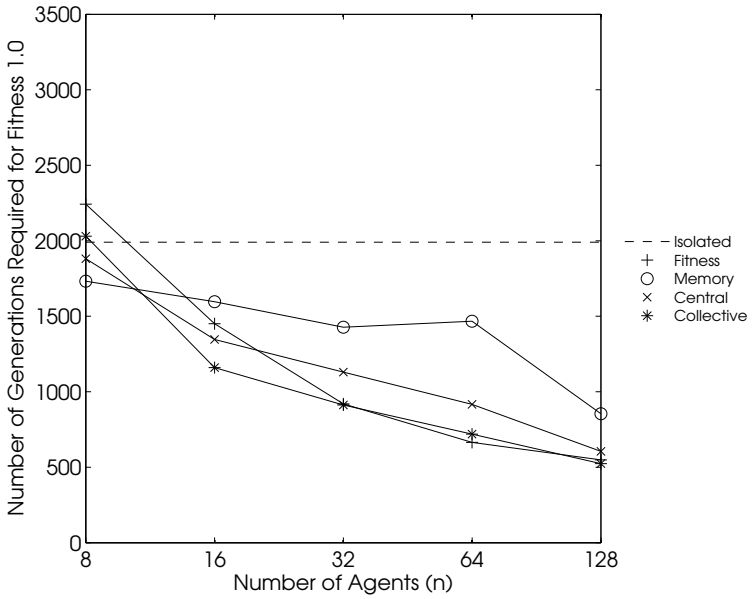


Figure 6.5: Speed performance of the three reputation-based models versus the fitness-based model.

better for $n = 8$ ($p < 0.01$).

The "collective" model performed better than the other two reputation models for $8 < n \leq 128$ ($0.01 < p < 0.08$). It also proved to be slightly but consistently ($p < 0.05$) better than the fitness-based model for all values of n, except for $n = 64$ where it performed roughly the same ($p > 0.5$).

Two surprising findings occur from these results:

- First, although we expected the "central" reputation model to be superior to the "collective" one, this did not prove to be the case. We can attribute this to the fact that reputation data is cumulative over time for the "collective" model (as opposed to fresh data aggregated from multiple agents for the "collective" model), which means that this particular algorithm is slower to change - i.e. more time is needed for an agent ruined by a defective peer to appear as such in the centralised database.

- A finding which is harder to explain is the fact that the "collective" model performs even better than the fitness-based model. Even though the difference in performance is small, the U-tests showed that it is pretty consistent. One explanation is that this difference may be attributable to the different selection pressures of the schemes used by the agents for mate selection (roulette wheel - high pressure for the fitness-based model, and tournament selection ($s = 2$ for

$d = 0$) - low pressure for the "collective" model), something which is supported to some extent by the connectivity results in section 5.3.6. However, this still does not make a convincing case for how any advantage offered by a reduced selection pressure would exceed the benefit of relying on a complete picture of current fitnesses as opposed to past experiences, especially at the absence of noise. Another explanation, perhaps more likely, is that the "collective" model allows agents to prefer peers with proven better mutation rates, something that the fitness-based model, relying solely on current population fitness, is unable to do.

6.3.3 Connectivity

When experimenting with the "collective" reputation-based model, we examined the impact of different levels of connectivity k on performance. As we did for the fitness-based model in Section 5.3.6, we tried five different values for k: Ask one, two, 25%, 50% and 100% of peers. The results for all five cases, for different agent population sizes, are given in Table 6.2. A visual representation of the results is given in Figure 6.6.

k	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
1	1837 *(496)*	1256 *(451)*	1150 *(422)*	1034 *(404)*	785 *(257)*
2	1699 *(472)*	1277 *(423)*	1177 *(619)*	1051 *(620)*	700 *(183)*
25%	1699 *(472)*	1191 *(282)*	1020 *(496)*	826 *(264)*	578 *(90)*
50%	1959 *(590)*	1209 *(364)*	897 *(295)*	714 *(164)*	691 *(172)*
100%	2026 *(509)*	1163 *(306)*	912 *(310)*	719 *(204)*	524 *(87)*

Table 6.2: Collective reputation model (no defective agents): Connectivity results. Averaged values (σ in parentheses).

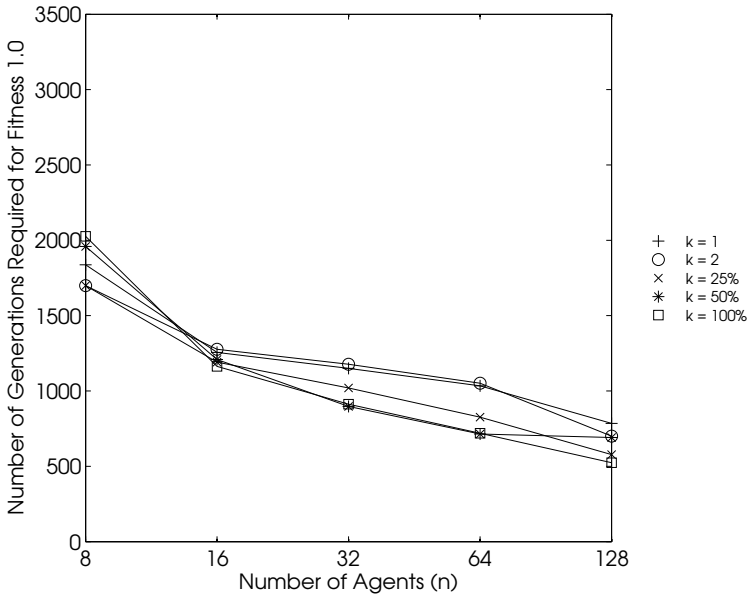


Figure 6.6: Collective reputation model (no defective agents): Connectivity results.

From these results we can see that for all agent population sizes, asking one peer is statistically equivalent to asking two ($p > 0.5$ in all cases). For $n > 16$, asking all peers performs better than asking just one or two ($p < 0.01$ in all cases). In most other cases, the number of peers asked is not statistically significant ($p > 0.05$).

6.3.4 Connectivity Under Noise

The introduction of a single defective agent did have an impact on the connectivity results presented in the previous section.

By looking at Table 6.3 and Figure 6.7, we can immediately see that the "collective" model manages to converge to the target fitness for all values of k tried, as well as all agent population sizes n.

As the cases of $k = 1$ and $k = 2$ show, keeping the number of suggesters k proportional to the total agent population size n helps maintain the scalability of this model.

Except for $n = 64$ ($p < 0.05$), asking 50% of peers was not significantly better than asking 25% ($0.1 < p < 0.5$).

Asking all peers produced significantly better performance for all values of n ($p < 0.05$), except for $n = 8$ where the difference in performance was reversed, but statistically insignificant ($p > 0.45$).

Regarding the scalability of the system, this is significantly maintained for $k = 50\%$ and $k = 100\%$ ($p < 0.05$ and $p < 0.01$ respectively), and a bit less consistently so for

$k = 25\%$ ($0.01 < p < 0.15$).

From these results we can infer that, in the presence of noise, the level of connectivity k does affect the performance of the system, in the expected way - i.e. asking a lot of peers yields better results than asking few.

k	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
1	2184 *(641)*	2686 *(903)*	2785 *(1109)*	3093 *(1037)*	2779 *(1336)*
2	2311 *(671)*	2253 *(685)*	2819 *(1036)*	3125 *(1140)*	2202 *(1012)*
25%	2311 *(671)*	1911 *(598)*	1586 *(667)*	1409 *(648)*	760 *(295)*
50%	2070 *(694)*	1837 *(716)*	1417 *(492)*	1064 *(539)*	785 *(277)*
100%	2230 *(771)*	1458 *(407)*	1044 *(394)*	751 *(177)*	563 *(144)*

Table 6.3: Collective reputation model (one defective agent): Connectivity results. Averaged values (σ in parentheses).

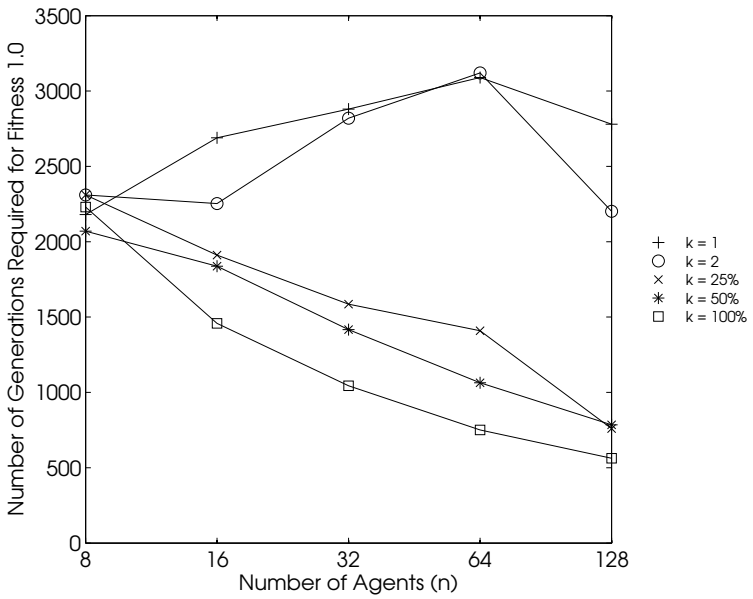


Figure 6.7: Collective reputation model (one defective agent): Connectivity results.

6.3.5 Relative Noise Tolerance

As a final comparison experiment, we performed runs of all three reputation-based models and contrasted their performance among them, as well as against the fitness-based model (with $k = 2$), for increasing levels of noise (from 0 to 50%). In the case of the "collective" reputation model, we compare two versions: One with $k = 2$ and one with $k = 50\%$. The former was included for comparison with the $k = 2$ fitness-based model, since these two have similar bandwidth requirements.

The results are given in Table 6.4 and Figure 6.8.

Model	$d = 0$	$d = 1$	$d = 2$	$d = 4$	$d = 8$	$d = 16$	$d = 32$	$d = 64$
Fitness ($k = 2$)	533 *(51)*	504 *(36)*	496 *(34)*	599 *(52)*	1213 *(203)*	4370 *1160*		
Memory	854 *(508)*	4164 *(1710)*	4280 *(1805)*	6073 *(1400)*	5975 *(1481)*	5763 *(1520)*	4504 *1751*	
Central	605 *(135)*	684 *(176)*	700 *(162)*	763 *(185)*	934 *(279)*	1165 *(384)*	1942 *582*	
Collective ($k = 2$)	700 *(183)*	2203 *(1013)*	3182 *(1134)*	4354 *(1294)*	4874 *(1281)*	5513 *(1234)*	5520 *1388*	6150 *(1255)*
Collective ($k = 50\%$)	691 *(172)*	785 *(277)*	736 *(183)*	726 *(155)*	734 *(120)*	962 *(233)*	1291 *346*	3315 *(1436)*

Table 6.4: Relative noise tolerance for all models ($n = 128$). Averaged values (σ in parentheses).

The fitness model offers good performance for low levels of noise (the best for $1 \leq d \leq 4$, $p < 0.05$), but breaks down fast for $d \geq 8$ where it performs significantly worse than the 50%-collective and the central models ($p < 0.01$). It failed to reach the target fitness for $d > 16$.

Regarding the reputation-based models, the "memory" algorithm is the worst performer in terms of absolute performance except for $d = 32$ ($p < 0.05$ in all cases), although it does have the advantage over the fitness-based model of being able to cope with higher levels of noise (up to $d = 32$ as opposed to $d = 16$ for the fitness-based model).

The "central" reputation algorithm offers significantly better performance for higher

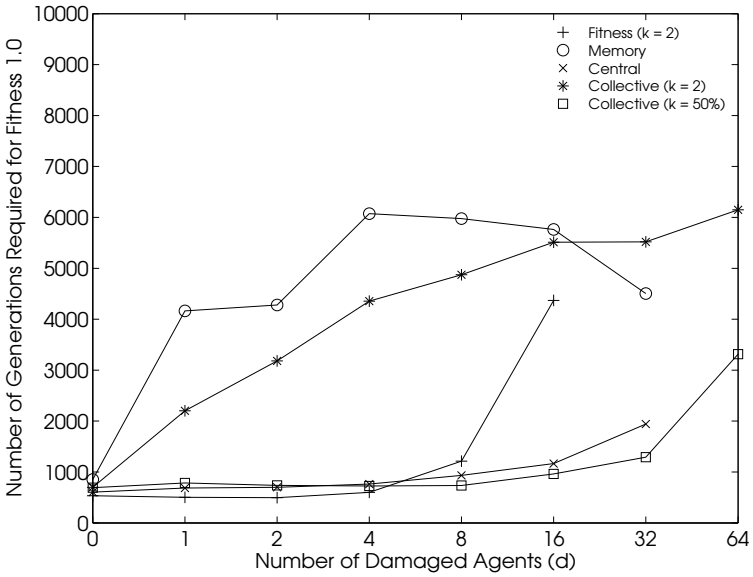


Figure 6.8: Relative noise tolerance for all models ($n = 128$).

levels of noise ($d \geq 8$) than the fitness-based model ($p < 0.01$), and seems to be affected less by increasing numbers of defective agents - all the way up to $d = 32$, after which it fails to converge. This failure can probably be attributed to the fact that, for too many defective agents ($d \geq 50\%$), the centralised database contains at least as much wrong reputation information (reported by the defective agents) as it does right.

The $k = 50\%$ version of the "collective" model behaves very similarly with the "central" model, offering slightly but consistently ($p < 0.05$) better performance than the latter for $d > 4$ - again, probably because it does not have to deal with as much wrong reputation information reported by defective peers. More importantly, it manages to cope gracefully with increasing levels of noise, all the way up to the maximum of $d = 64$ - i.e. where half of all agents were defective.

The $k = 2$ version of the "collective" model also manages to deal with very high levels of noise (again up to the maximum of $d = 64$), although - as expected based on the results in section 6.3.4 - with significantly inferior performance than its $k = 50\%$ version or the "central" algorithm ($p < 0.01$ for $d > 0$). Comparing it with the fitness-based model, we can infer that for the same bandwidth the latter offers significantly better results for $d \leq 8$, but worse-to-failure beyond this ($p < 0.01$ in all cases).

6.3.6 Noise Profile

For our final experiment, we generated a noise profile for the "collective" reputation model (for $k = 50\%$), similar in scope to the one presented in section 5.3.8.

Again, we performed runs using 8, 16, 32, 64 and 128 agents, and in each case we introduced an increasing number of defective agents (1, 2, 4, 8, 16, 32 and 64, up to a maximum of 50% in each agent configuration). Each of the 40 averaged runs of each permutation was allowed to reach 10,000 generations, and the best (minimum), final fitness of the best performing agent in each case was recorded.

The results are illustrated in Figure 6.9, with the actual results given in Table 6.5.

Contrasting this graph with the corresponding one for the fitness-based model (Figure 5.14) allows us to immediately see the improved noise tolerance of the reputation-based model. Where the fitness-based model broke down after about 3% damage (for $n \geq 32$), in the case of the reputation-based model, the break-down point occurs at about 25% damage.

In addition, even for very high levels of noise (50%), the reputation-based model manages to converge to a fitness several orders of magnitude better than the fitness-based model, and more so for larger agent populations - unlike the fitness-based model, where high levels of noise also compromised the system's scalability.

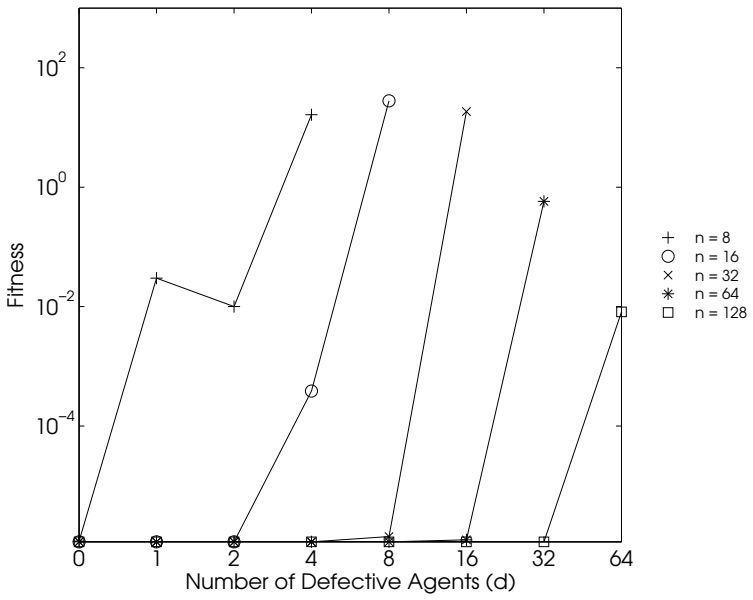


Figure 6.9: Noise profile for the "collective" algorithm ($k = 50\%$).

Defective	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
0	1.18E-6 *(1.44E-7)*	1.15E-6 *(0)*	1.15E-6 *(0)*	1.15E-6 *(0)*	1.15E-6 *(0)*
1	0.03 *(0.16)*	1.15E-6 *(0)*	1.15E-6 *(0)*	1.15E-6 *(0)*	1.15E-6 *(0)*
2	0.01 *(0.06)*	1.15E-6 *(0)*	1.15E-6 *(0)*	1.15E-6 *(0)*	1.15E-6 *(0)*
4	16.33 *(48.08)*	3.84E-4 *(1.89E-3)*	1.15E-6 *(0)*	1.15E-6 *(0)*	1.15E-6 *(0)*
8		27.99 *(50.40)*	1.41E-6 *(1.59E-6)*	1.15E-6 *(0)*	1.15E-6 *(0)*
16			18.48 *(35.97)*	1.24E-6 *(5.34E-7)*	1.15E-6 *(0)*
32				0.58 *(1.90)*	1.15E-6 *(0)*
64					8.17E-3 *(3.33E-2)*

Table 6.5: Noise profile for the “collective” algorithm ($k = 50\%$).

6.4 Discussion

We have shown that it is possible to use a reputation mechanism, as used in electronic institutions, as an adequate fitness indicator in an evolutionary multi-agent system.

The performance of this approach in our experiments was found to be similar or better (depending on the scope and connectivity of the reputation model used) to that of direct fitness observation in the absence of noise. When defective agents were added in the network, the reputation-based models offered improved tolerance to noise, while maintaining their scalability for increasing agent population sizes.

Different reputation models offered different performance and different benefits for different costs. In general, increasing the connectivity between agents (for the “collective” model) improved noise tolerance, although at the cost of additional bandwidth.

Finally, the non-centralised reputation models (“memory” and “collective”) were

found to be less susceptible to misleading reputation information reported by defective agents.

6.5 Summary

In this chapter, we have discussed the following points:

- A description of our reputation models and how these were used as the selection bias in the evolution of the agents as a replacement for the traditional approach of direct fitness observation.
- The setup and parameters used for our experiments.
- Results from a series of experiments, showing:

  How the reputation-based models manage to cope with noise in the form of defective agents.

  A comparison in terms of performance of the three reputation-based algorithms versus our benchmarks, showing that the former perform similarly or even better than the fitness-based model presented in Chapter 5.

  An investigation into the effect of different levels of connectivity on the performance of the reputation-based algorithm, which indicates that increased connectivity is not necessary for this algorithm to perform well in the absence of noise, but does seem to help when defective agents are introduced.

  A noise profile for the "collective" reputation model, illustrating the relative performance of varying agent population sizes operating under varying levels of noise (up to 50%).

Chapter 7

Conclusion

7.1 Contributions

In summary, the main contributions that our investigation has made to the combined fields of Multi-agent Systems and Evolutionary Computation are the following:

7.1.1 The LiJ Interpreter

In Chapter 3, we presented the general architecture and implementation details of the LiJ interpreter, which is capable of executing coordinated multi-agent interactions specified in the LCC language, with constraints defined in Java. LiJ served as the basis of our experimentation software platform throughout our research.

Although the idea of a Java-based LCC interpreter is not new in itself, our particular implementation has the advantages of being fast, easy to use and automate, and also very reliable - even in configurations involving large numbers of agents and relatively complex interaction protocols. These advantages were critical for conducting our experiments successfully and within practical time limits.

Its main limitation, albeit a deliberate one, is its lack of network support, which was left out in order to reduce the overall execution overhead and allow our experiments to run faster on single-machine and/or cluster computation environments. Even so, LiJ's clean architecture offers easy extensibility, thus allowing network support to be added in the future if so required.

7.1.2 A P2P Parallel Adaptive GA

In Chapter 5 we described how the LiJ interpreter was used to design and deploy a multi-agent system specified in the LCC language, that can act as a distributed, decentralised (P2P) genetic algorithm capable of adaptation.

The results from our experiments with this first version of our algorithm, detailed in Section 5.3, confirmed our expectations of good scalability characteristics which, when the algorithm is deployed on parallel computational systems such as multi-core SMP processors, clusters and (in theory) networks, results in improved performance - in terms of the number of generations required by any one node for a given target fitness.

In addition, its adaptive properties aim to remove the burden of parameterisation from the user, making it ideal for use by people with little experience on evolutionary computation, as well as in situations where the fitness function being optimised changes dynamically (although this last point is speculative, as such a scenario has not been tested yet).

7.1.3 A Reputation-based Evolutionary MAS

In Chapter 6 we documented the next, and final, step towards our objective of a noise-tolerant, P2P evolutionary MAS. We showed how the first version of our algorithm was modified by replacing direct fitness observation (from the point of view of the individual peers in the MAS) by a simple reputation model.

The experimental results presented in Section 6.3 illustrate how this reputation-based variation of our algorithm maintains a high performance and scalability, statistically identical to the first, fitness-based version, while at the same time benefiting from the intrinsic characteristics of reputation: The exploitation of the common, collective experiences of all of the agents in the system (as opposed to the fitness-based version, where each agent can only rely on its own individual observations), which in turns results in an improved tolerance to faulty agents - a common scenario in open distributed systems.

Faulty agents do incur a cost in performance, however this cost remains directly proportional to the ratio of defective agents in the population, which implies that our architecture maintains its scalability in large, open computation environments, even at the presence of noise.

7.2 Future Work

7.2.1 Extending The LiJ Interpreter

Adding Network Support

As mentioned in Section 3.2.3, the LiJ interpreter was deliberately built without network support, since during our research we were only interested in running experiments in isolated, multi-core computers. A real-world multi-agent system, however, shows its true power in larger scale computational environments, such as large networks, grid infrastructures etc. This necessitates support for networking capabilities.

While designing the interpreter, we kept this (future) requirement in mind, and the resulting architecture reflects this: Adding network support to LiJ requires a number of straightforward, yet non-trivial alterations.

As it stands, an LCC interaction model (IM) executed with the current version of LiJ involves multiple threads, one for each agent, as well as a single, common runtime process, which is responsible for the following:

- Protocol loading.
- Agent subscriptions.
- Handling of message passing between agents.
- Providing information to agents about their peers.

In a networked implementation this functionality would have to be distributed across the network, with each agent (or set of agents residing on the same machine) requiring a separate runtime.

The individual LiJ runtimes spread across the network would require a mechanism for discovering and communicating with each other. The OpenKnowledge framework achieves this by means of a *Discovery Service*, a custom-built, centralised service capable of publishing IMs and handling agent subscriptions to the available IMs, among other things.

A similar approach can be used for LiJ, with a central server on which each LiJ instance can connect to in order to coordinate with other, remote LiJ instances. An alternative approach, more faithful to the peer-to-peer, decentralised paradigm, would be for each LiJ instance to incorporate an individual service capable of storing and/or discovering similar services across the network and connecting to them.

From the point of view of a user deploying an IM, LCC protocols would look the same: Constraint method arguments in LiJ are already declared as Java *Serializable* objects for that purpose. Making use of Java's RMI registry and framework for implementing the network component (a single centralized one or multiple individual ones in each of the LiJ instances) would mean that no further work is required on our part for flow control and marshalling objects.

Automatic Identifier Allocation

In its current implementation, the LiJ interpreter requires each subscribing agent to provide its own identifier/ID. This was done in order to facilitate debugging and experimentation. In real-world applications, however, where security is an issue, this scheme is not adequate as it is prone to spoofing - i.e. malicious agents claiming IDs other than their own.

Enabling automatic ID allocation by the interpreter is a simple and straightforward modification. As it is easy for a particular agent to discover its own ID, interactions requiring specific agent instances can be implemented by having this ID information exchanged between the participating agents as part of the interaction model being executed.

Dealing with Infinite Recursion

In Section 3.3.5 we discussed how we solved the problem of stack overflows in the Java VM caused by infinite (or simply lengthy) tail-recursion, by introducing an additional role type, *cyclic*. Despite this work-around working perfectly well for our purposes, it does have the drawback of adding a bit more complexity to the process of writing LCC protocols.

As an alternative to this quick-and-dirty approach, we can implement proper flow analysis and tail-recursion elimination in LiJ (using techniques described in, e.g., [Muchnick, 1997]), and thus produce a more elegant, "smarter" interpreter.

7.2.2 Adaptive P2P GA

Complete GA Adaptation

As stated in Section 5.2.1, and in order to simplify our investigation, during our experiments with the adaptive parallel GA only the mutation rate was allowed to adapt. This

is of course not very effective for a real-life application, where the full range of genetic algorithm parameters (population size, elite size, crossover rate, selection strategy etc.) needs to be adapted as the evolutionary process progresses.

This extension is relatively straightforward to implement, as the basic characteristics of the architecture's implementation remain unaffected.

Asynchronous Agent Operation

Currently, all agents in our system work synchronously. This means that they all perform the same number of iterations before every extra-agent crossover stage, with faster agents having to wait for the slower ones to catch up.

When the system is deployed in a network consisting of computational elements of similar capabilities, this strategy works fine. However, in networks with diversified computational elements, this scheme is obviously inefficient.

The current coordination protocol could be modified in order to resolve this, by using time- or fitness-based cycle lengths rather than generation-based ones, and allowing the agents to handle peer requests in parallel with the execution of their intra-agent GA.

Additional Solvers

Finally, it will be interesting to take full advantage of the openness inherent to our architecture and LCC, by allowing additional kinds of solvers to be introduced in the system in addition to our standard GA (e.g. gradient search, simulated annealing, etc.). This will require the re-design of our protocol regarding the extra-agent crossover, or possibly the co-existence of more than one protocol in the system.

We believe that the effort required will be justified, since, by extending our architecture in this way, we will effectively be creating an open, peer-to-peer, adaptive hybrid optimisation platform.

7.2.3 Reputation-based Algorithm

Even More Noise

The nature of the noisy agents used in the experiments presented in Chapter 6 is rather simplistic, in the sense that all "defective" agents behave in an identical way. As a result, it is easy to envision a system much simpler than the one we discussed, that

would be able to defend against this kind of (predictable) behaviour equally well, if not even better.

In the future, it will be interesting to experiment with different forms, or extent, of damage - i.e. purely random-damage agents, deliberately lying agents, slandering, etc. - as well as rotating the "damage" among agents.

Alternative Reputation Models

A next step in taking this research further would be to implement and test alternative reputation models, especially in relation to different noise configurations as mentioned above.

As the literature on probabilistic as well as more complex, heuristic- or inference-based reputation mechanisms and the resulting emergent behaviour is constantly being enriched with further research, it will be interesting to see how some of these perform in our particular case.

7.2.4 Towards a Generic, Self-optimizing MAS Platform

In Section 5.2.2, we referred to our architecture's ability to "optimise the optimiser itself". We can extend this paradigm to a more general case, and allow the algorithm (whether it be the fitness- or the reputation-based one) to optimise, not GA solver agents exclusively, but other types of agent as well.

Provided that the process performed by an agent can be parameterised and expressed as a genome, and as long as the agent can maintain a measure of its fitness, we can have a group of agents perform any such parameterisable process in parallel, and exploit the evolutionary and adaptive properties of our algorithm to optimise the agents in real time, as well as allow them to adapt to change in dynamic environments. And all this would emerge as a direct consequence of the evolutionary nature of the system, without any modification to the agents' process itself.

The nature of the process matters little, from the point of view of the algorithm. It might be the case that each agent contains an Artificial Neural Network (ANN), or a functional mathematical model. The task at hand may be driving an autonomous robot, or predicting prices in the stock market. As long as the agents involved are homogeneous, and thus have compatible genes that can be recombined, evolution should take care of the rest.

7.3 Epilogue

The motivation for this thesis, together with the list of goals that we set when we began, both discussed in Chapter 1, can be combined to form what, during the course of our research, progressively became our vision: The theoretical grounding and practical implementation of an open, P2P multi-agent architecture, wherein agents are evolving, adaptive social entities rather than static performers.

Although the practical realisation of this vision is still far, we believe that our research has brought us one step closer to it.

Appendix A

LCC Reference Manual

A.1 Syntax Specification

The formal specification of the LCC language is given in Figure A.1 below:

$$
\begin{aligned}
Framework &:= Clause, \ldots \\
Clause &:= Role :: Def \\
Role &:= a(Type, Id) \\
Def &:= Role \mid Message \mid Null \mid Def\ then\ Def \mid Def\ or\ Def \leftarrow C \\
Message &:= M \Rightarrow Role \mid M \Leftarrow Role \\
C &:= Term \mid C \wedge C \mid C \vee C \\
M &:= Term \\
Type &:= Term \\
Id &:= Constant \mid Variable \\
Term &:= Constant \mid Variable \mid P(Term, \ldots)
\end{aligned}
$$

Figure A.1: LCC syntax specification.

A.2 User Guide

A.2.1 Introduction

As its name suggests, LCC was designed to be a lightweight language. An attempt to describe it in a single sentence would result in something like this:

> An interaction model expressed as an LCC protocol consists of a series of definitions, or *defs*, separated by *then* and *or* operators and guarded by *constraints*, grouped into *role* clauses.

In the sections that follow, we describe these primitives in more detail, and give examples on how they can be used to implement interaction models between multiple agents.

A.2.2 Comments

The LiJ interpreter supports both single-line and multi-line comments inside protocol source files.

Comments are given in the C-style. A single-line comment (C99-style) can be located anywhere on a line, starts with a double-slash, and ends at the end of that line. A multi-line comment (C89-style) can be located anywhere in the source file, starts with a slash-star, and ends with a star-slash.

```
\\ A single-line comment


\*
    A multi-line
    comment
*\
```

A.2.3 Roles

At any given time, an agent participating in an interaction model assumes a role, with each role being defined in a clause.

Each role requires a role declaration to be given at the beginning of the protocol source file. Role declarations are given in the following way:

```
r(roleName, roleType, args)
```

A *roleName* can be any alphanumeric constant beginning with a lower-case letter, or a term (in the Prolog sense) consisting of such an alphanumeric constant along with comma-separated arguments inside parentheses.

The *roleType* specifies the nature of the role being declared, and can be one of the following:

- `initial`

  This is the first role that will be executed by the interpreter when the interaction model stars. Exactly one such role is required in an interaction model, and only a single agent can subscribe to it. No *args* parameter is specified for this role type.

- `necessary`

  Interaction models containing roles declared with this type will require at least as many agent subscribers to each such role as specified in the provided *args* value.

- `optional`

  Subscription of agents to roles of this type is not necessary for an interaction model to commence, but such agents can still participate if present. No *args* parameter is specified for this role type.

- `auxiliary`

  Roles of this type are usually not subscribed to by agents directly. Instead, they are switched to by agents during runtime, and serve to temporarily change their behaviour. No *args* parameter is specified for this role type.

- `cyclic`

  This special role type indicates a clause that will be automatically looped by the interpreter. It is similar to the *auxiliary* type in scope.

A.2.4 Clauses

For each role declaration specified at the top of the protocol source file, there must be an identically-termed clause definition present. A clause is defined in the following way:

```
a(roleName, ID}::Def
```

The *roleName* term must match the one given in the corresponding role declaration. The *ID* variable contains the identifier of the agent assuming that role. *Defs* are discussed in the following section.

A.2.5 Defs

A def can be one of the following:

- *Role*
  This def will cause the agent to switch to the role specified by the *Role* term. Data can be passed along between roles as part of the term's arguments.

- *Message*
  A *Message* def will send/receive a message matching the specified *Message* term to/from the agent matching the specified role term and ID. The underscore wildcard operator, _, can be used instead of an ID when the identity of the agent is irrelevant.

- *Null*
  This def will perform no action. It is typically used as a placeholder for constraints, or as a default fallback for when a list iteration finishes (see Section A.2.9 below).

A.2.6 Constraints

A constraint acts as a guard for the execution of the associated def. A constraint can evaluate to either *true* or *false* (in the LiJ interpreter there is a third possibility, the *maybe* state - see Section 3.3.4 for more information). Unless the constraint evaluates to *true*, the guarded def will not be executed.

Constraints are appended at the end of a def, and are denoted by the arrow operator:

```
Def <-- Constraint
```

Constraints can be of one of the following types:

- *Comparison*
  A *Comparison* constraint will perform a comparison between two constant and/or variable values. The operators supported by LiJ are:

  - *Less than* (operator: $<$)
  - *Greater than* (operator: $>$)
  - *Equal to* (operator: $==$)
  - *Not equal to* (operator: $!=$)

- *Assignment*
  An *Assignment* constraint is not strictly a constraint, as it always evaluates to *true*. Its purpose it to assign values to variables, by using the standard assignment ($=$) operator.

- *Method*
  A *Method* constraint is a direct call to a Java method defined by the subscribing agent executing the current clause (see Section A.2.10 below for more information on how to provide these). It will evaluate to whatever result that method returns. As mentioned above, three logic states are supported by LiJ: *true*, *false* and *maybe*. This last state can be exploited by an agent designer to "suspend" execution in a non-blocking way (by returning *maybe*), until a latter time - e.g. until some user input.

Multiple constraints can be combined using the logical operators *and* and *or*.

A.2.7 Sequence and Choice

LCC defs are joined to each other using one of two possible operators: *then* and *or*.

- `then`

  A set of defs joined with *then* operators will evaluate to *true* if and only if every individual def in the group evaluates to *true*.

- `or`

  A set of defs joined with *or* operators will evaluate to *true* if at least one individual def in the group evaluates to *true*.

In both cases, defs are evaluated in the order they are encountered. For the complete tri-state truth tables of these operators, please refer to Section 3.3.4.

A.2.8 Data Types

At the moment, the LiJ interpreter supports the following data types for constants and variable values:

- Integer numbers
- Floating-point numbers
- String literals (enclosed in double-quotes)

Values passed via constraint method arguments, however, can be of any Java class implementing the *Serializable* interface.

A.2.9 Lists and Recursion

Lists are an important part of LCC. Using lists, it is possible to iterate through a number of values using recursion.

A list in LCC has the following form:

```
L = [H | T]
```

L denotes the list structure, *H* denotes the head of the list (i.e. the first element in it), and *T* denotes the tail of the list (i.e. the remaining elements in it).

Depending on the contents of the head variable *H*, this same assignment can be used to either append an element to the list, or extract one from it:

- If *H* contains a value, this assignment will append that value to the list *L*.
- If *H* is empty, then the assignment will extract the first element (the head) of *L* into it, and will store the remaining elements of *L* into *T*.

By using the second case, it is possible to iterate through all of the elements in a list, using the tail *T* (itself a list) for recursion. When the list *L* becomes empty, the assignment will fail.

To test whether a list is empty, we can use the standard comparison constraint, $==$, as follows:

```
L == []
```

A.2.10 Java Method Constraints

When subscribing an agent for participation in an interaction model using the LiJ interpreter (using the *subscribe* method of the *lij.runtime.Interpreter* class), we need to specify three things:

- The name of the role for which the agent subscribes.
- An identifier for the agent.
- A Java object implementing the *lij.interfaces.ConstraintImplementor* interface.

As its name implies, a *ConstraintImplementor* object defines and supplies methods that can be used as constraints. The way in which such methods work are not important from the point of view of the LiJ interpreter. The only things that are required from a constraint method are:

- It must accept arguments that implement the *lij.interfaces.Accessor* interface.
- It must return either a *boolean* value, or - for tri-state logic - one of the three states specified in the *lij.interfaces.Result* class (*State.TRUE*, *State.FALSE* or *State.MAYBE*).

A.2.11 LiJ Special Constraints

Special constraints are Java constraint methods that do not have to be provided by an agent's *ConstraintImplementor*, but are instead provided by the LiJ interpreter itself to all agents. At the moment, LiJ provides only one such method, which has the signature *_findPeers(Accessor role, Accessor list)*.

The *_findPeers* special constraint method can be used by an agent to discover the IDs of peers that perform a certain role. The *role* argument must contain the name of a role in the interaction model, while the *list* argument, which is a return argument, will contain a list structure with the IDs of all the agents in the interaction that match the specified role (excluding the ID of the calling agent). If no role name is specified, the IDs of all agents in the interaction (excluding the calling agent's) will be returned.

The *_findPeers* constraint method will evaluate to *true*, as long as at least one matching agent ID is found by the interpreter.

A.3 Examples

A.3.1 Hello World

LCC Protocol

```
/**
  A very simple interaction, with two peers: a greeter and a
     responder.
*/



r(greeter, initial)
r(responder, necessary, 1)



a(greeter, ID1) ::
    greeting(Message) => a(responder, ID2) <- userInitiatesGreeting(
       Message)
    then
    response(Reply) <= a(responder, ID2)
    then
    null <- displayReply(Reply)
    then
    a(greeter, ID1)



a(responder, ID2) ::
    greeting(Message) <= a(greeter, ID1)
    then
    response(Reply) => a(greeter, ID1) <- createReply(Message, Reply
       )
    then
    a(responder, ID2)
```

Java Source Code

```
import java.io.File;
import java.io.FileInputStream;
import javax.swing.JOptionPane;
import lij.runtime.Interpreter;
import lij.exceptions.InterpreterException;
import lij.interfaces.Accessor;
import lij.interfaces.ConstraintImplementor;



public class Main
{
  private static final File INPUT_FILE = new File("./helloworld.lcc"
     );



  public static void main(String[] args)
  {
```

```
    try
    {
      // Create interpreter
      Interpreter interpreter = new Interpreter(new FileInputStream(
          INPUT_FILE), false);

      // Subscribe agents
      interpreter.subscribe("greeter", new AgentGreeter());
      interpreter.subscribe("responder", new AgentResponder());

      // Run IM
      interpreter.run();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}



public class AgentGreeter implements ConstraintImplementor
{
  public boolean userInitiatesGreeting(Accessor Message) throws
      InterpreterException
  {
    String input = JOptionPane.showInputDialog("Greeting?");
    if (input == null)
      return false;

    Message.setValue(input);

    return true;
  }



  public boolean displayReply(Accessor Reply)
  {
    JOptionPane.showMessageDialog(null, Reply.getValue());

    return true;
  }
}



public class AgentResponder implements ConstraintImplementor
{
  public boolean createReply(Accessor Message, Accessor Reply)
      throws InterpreterException
  {
    Reply.setValue(Message.getValue() + ", world..!");

    return true;
  }
}
```

A.3.2 Ping

LCC Protocol

```
/**
  Demonstrates the _findPeers special constraint,
  as well as the use of a tri-state constraints (using a button for
     input).
*/



r(agent, initial)
r(pinger(S), auxiliary)



a(agent, A) ::
  (
    (
      null <- waitForClick()
      then
      null <- _findPeers("agent", S)
      then
      (
        a(pinger(S), A)
        or
        a(agent, A)
      )
    )
    or
    (
      ping() <= a(pinger, X)
      then
      null <- gotPing(X)
    )
  )
  then
    a(agent, A)



a(pinger(S), A) ::
  null <- S = [H | Sr]
  then
    ping() => a(agent, H)
    then
    a(pinger(Sr), A)
```

Java Source Code

```
import java.awt.BorderLayout;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.FileInputStream;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import lij.exceptions.InterpreterException;
```

```
import lij.runtime.Interpreter;
import lij.interfaces.Accessor;
import lij.interfaces.ConstraintImplementor;
import lij.interfaces.Result;



public class Main
{
  private static int id = 0;
  private static final File INPUT_FILE = new File("./ping.lcc");



  public static void main(String[] args)
  {
    try
    {
      // Create interpreter
      final Interpreter interpreter = new Interpreter(new
         FileInputStream(INPUT_FILE), false);

      // Setup frame
      JFrame f = new JFrame("Main");
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      f.setBounds(0, 0, 240, 120);
      final JButton b = new JButton("Add Agent " + id);
      f.getContentPane().add(b);
      f.setVisible(true);
      b.addActionListener(new ActionListener()
      {
        public void actionPerformed(ActionEvent ae)
        {
          try
          {
            interpreter.subscribe("agent", new Agent(id), id);
            id++;
            b.setText("Add Agent " + id);
          }
          catch (InterpreterException e)
          {
            e.printStackTrace();
          }
        }
      });

      // Run IM
      interpreter.run();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}



public class Agent implements ConstraintImplementor, ActionListener
{
  public static final int SCREEN_WIDTH = (int)Toolkit.
     getDefaultToolkit().getScreenSize().getWidth();
```

```
public static final int SCREEN_HEIGHT = (int)Toolkit.
    getDefaultToolkit().getScreenSize().getHeight();
public static final int FRAME_WIDTH = 240;
public static final int FRAME_HEIGHT = 120;
private static int lastX = 0;
private static int lastY = 120;
private int id;
private JButton b = new JButton("Ping");
private JLabel l = new JLabel(" ");
private boolean buttonPressed = false;



public Agent(int _id)
{
  id = _id;

  b.addActionListener(this);

  JFrame f = new JFrame("Agent " + id);
  f.setBounds(lastX, lastY, FRAME_WIDTH, FRAME_HEIGHT);
  lastX += FRAME_WIDTH;
  if (lastX + FRAME_WIDTH > SCREEN_WIDTH)
  {
    lastX = 0;
    lastY += FRAME_HEIGHT;
  }
  f.getContentPane().add(b, BorderLayout.CENTER);
  f.getContentPane().add(l, BorderLayout.SOUTH);
  f.setVisible(true);
}



public void actionPerformed(ActionEvent ae)
{
  buttonPressed = true;
}



public Result.State waitForClick()
{
  if (!buttonPressed)
  {
    return Result.State.MAYBE;
  }
  else
  {
    buttonPressed = false;
    return Result.State.TRUE;
  }
}



public Result.State gotPing(Accessor ID)
{
  l.setText("Got ping from: " + ID.getValue().toString());

  return Result.State.TRUE;
}
```

```
}
```

A.3.3 Dining Philosophers

LCC Protocol

```
/**
  An LCC implementation of the Dining Philosophers paradigm.
*/



r(waiter, initial)
r(waiter(L), auxiliary)
r(philosopher, necessary, 5)



a(waiter, I) ::
  a(waiter([0, 1, 2, 3, 4]), I)



a(waiter(L), W) ::
  a(waiter, W) <- L==[]
  or
  (
    null <- L = [P | Lr]
    then
    prompt => a(philosopher, P)
    then
    (
      (
        requestFork(ForkIndex) <= a(philosopher,P)
        then
        (
          fork => a(philosopher, P) <- giveFork(ForkIndex)
          or
          wait => a(philosopher, P)
        )
      )
      or
      (
        returnForks(ReturnedForkIndexLeft, ReturnedForkIndexRight)
            <= a(philosopher, P)
        then
        null <- forkReturned(ReturnedForkIndexLeft) and forkReturned
            (ReturnedForkIndexRight)
      )
      or
      (
        requestNothing <= a(philosopher, P)
      )
    )
    then
    a(waiter(Lr), W)
  )
```

```
a(philosopher, P) ::
  prompt <= a(waiter, W)
  then
  null <- updateDesire()
  then
  (
    (
      requestFork(ForkIndex) => a(waiter(L), W) <- wantsFork(
         ForkIndex)
      then
      (
        (
          fork <= a(waiter(L), W)
          then
          null <- gotFork(ForkIndex)
        )
        or
        (
          wait <= a(waiter(L), W)
          then
          null <- gotWait(ForkIndex)
        )
      )
    )
    or
    (
      returnForks(ReturnedForkIndexLeft, ReturnedForkIndexRight) =>
         a(waiter(L), W) <- wantsStartThinking(ReturnedForkIndexLeft
         , ReturnedForkIndexRight)
    )
    or
    (
      requestNothing => a(waiter(L), W) <- wantsNothing()
    )
  )
  then
  a(philosopher, P)
```

Java Source Code

```
import java.io.File;
import java.io.FileInputStream;
import java.awt.Toolkit;
import java.awt.event.ComponentAdapter;
import java.awt.event.ComponentEvent;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Toolkit;
import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JLayeredPane;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.SwingConstants;
import lij.runtime.Interpreter;
import lij.exceptions.InterpreterException;
import lij.interfaces.Accessor;
import lij.interfaces.ConstraintImplementor;
```

```
public class Main
{
  private static final File INPUT_FILE = new File("./
      diningphilosophers.lcc");



  public static void main(String[] args)
  {
    try
    {
      // Create interpreter
      Interpreter interpreter = new Interpreter(new FileInputStream(
          INPUT_FILE), false);

      // Subscribe agents
      interpreter.subscribe("waiter", new AgentWaiter());
      interpreter.subscribe("philosopher", new AgentPhilosopher(0,
          0, 4), 0);
      interpreter.subscribe("philosopher", new AgentPhilosopher(1,
          1, 0), 1);
      interpreter.subscribe("philosopher", new AgentPhilosopher(2,
          2, 1), 2);
      interpreter.subscribe("philosopher", new AgentPhilosopher(3,
          3, 2), 3);
      interpreter.subscribe("philosopher", new AgentPhilosopher(4,
          4, 3), 4);


      // Run IM
      interpreter.run();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}



public class AgentWaiter implements ConstraintImplementor
{
  private static final int WIDTH = 320;
  private static final int HEIGHT = 340;
  private static final int SPAGHETTI_RADIUS = 100;
  private static final int FORK_RADIUS = 100;
  private static final ImageIcon ICON_SPAGHETTI = new ImageIcon("res
      /spaghetti.png");
  private static final int ICON_WIDTH = ICON_SPAGHETTI.getIconWidth
      ();
  private static final int ICON_HEIGHT = ICON_SPAGHETTI.
      getIconHeight();
  private JFrame frame = null;
  private JLayeredPane lPane = new JLayeredPane();
  private JLabel lTable = new JLabel(new ImageIcon("res/table.png"))
      ;
  private JLabel[] lSpaghetti = new JLabel[5];
  private JLabel[] lForks = new JLabel[5];
  public boolean[] forks = new boolean[] { true, true, true, true,
```

```
    true };


public AgentWaiter()
{
  // Set (up) the table...
  lPane.add(lTable, 1);
  for (int i = 0; i < 5; i++)
  {
    lSpaghetti[i] = new JLabel(ICON_SPAGHETTI);
    lSpaghetti[i].setOpaque(false);
    lPane.add(lSpaghetti[i], 0);
    lForks[i] = new JLabel(new ImageIcon("res/fork" + i + ".png"))
       ;
    lForks[i].setOpaque(false);
    lPane.add(lForks[i], 0);
  }

  // Frame location
  int centreX = (int)(Toolkit.getDefaultToolkit().getScreenSize().
     getWidth() / 2.0);
  int centreY = (int)(Toolkit.getDefaultToolkit().getScreenSize().
     getHeight() / 2.0);
  int x = centreX - WIDTH / 2;
  int y = centreY - HEIGHT / 2;

  // Setup GUI
  frame = new JFrame("Waiter");
  frame.setResizable(false);
  frame.setBounds(x, y, WIDTH, HEIGHT);
  frame.setContentPane(lPane);
  frame.addComponentListener(new ComponentAdapter()
  {
    public void componentShown(ComponentEvent ce)
    {
      int width = frame.getContentPane().getWidth();
      int height = frame.getContentPane().getHeight();
      lTable.setBounds(0, 0, width, height);
      for (int i = 0; i < 5; i++)
      {
        double spaghettiAngle = i * 2.0 * Math.PI / 5.0 - Math.PI
           / 2.0;
        int tableCentreX = width / 2;
        int tableCentreY = height / 2;

        // Spaghetti
        int spaghettiOffsetX = (int)(Math.cos(spaghettiAngle) *
           SPAGHETTI_RADIUS);
        int spaghettiOffsetY = (int)(Math.sin(spaghettiAngle) *
           SPAGHETTI_RADIUS);
        int spaghettiX = tableCentreX + spaghettiOffsetX -
           ICON_WIDTH / 2;
        int spaghettiY = tableCentreY + spaghettiOffsetY -
           ICON_HEIGHT / 2;
        lSpaghetti[i].setBounds(spaghettiX, spaghettiY, ICON_WIDTH
           , ICON_HEIGHT);

        // Forks
        double forkAngle = i * 2.0 * Math.PI / 5.0 - Math.PI / 2.0
            + Math.PI / 5.0;
        int forkOffsetX = (int)(Math.cos(forkAngle) * FORK_RADIUS)
```

```
            ;
          int forkOffsetY = (int)(Math.sin(forkAngle) * FORK_RADIUS)
            ;
          int forkX = tableCentreX + forkOffsetX - ICON_WIDTH / 2;
          int forkY = tableCentreY + forkOffsetY - ICON_HEIGHT / 2;
          lForks[i].setBounds(forkX, forkY, ICON_WIDTH, ICON_HEIGHT)
            ;
        }

        updateGUI();
      }
    });

    frame.setVisible(true);
  }



  public boolean giveFork(Accessor ForkIndex) throws
      InterpreterException
  {
    int forkIndex = (Integer)ForkIndex.getValue();

    boolean result = false;
    if (forks[forkIndex])
    {
      forks[forkIndex] = false;
      result = true;
    }

    updateGUI();

    return result;
  }



  public boolean forkReturned(Accessor ForkIndex) throws
      InterpreterException
  {
    int forkIndex = (Integer)ForkIndex.getValue();

    forks[forkIndex] = true;

    updateGUI();

    return true;
  }



  private void updateGUI()
  {
    for (int i = 0; i < 5; i++)
    {
      if (forks[i])
        lPane.add(lForks[i], 0);
      else
        lPane.remove(lForks[i]);
    }

    lPane.revalidate();
```

```
    lPane.repaint();
  }
}



public class AgentPhilosopher implements ConstraintImplementor
{
  private static final String[] NAMES = new String[] { "Plato", "
     Konfuzius", "Socrates", "Voltaire", "Descartes" };
  private static final String DESIRE_EAT = "EAT";
  private static final String DESIRE_THINK = "THINK";
  private static final String STATE_THINKING = "THINKING";
  private static final String STATE_EATING = "EATING";
  private static final String STATE_WAITING_LEFT = "WAITING_LEFT";
  private static final String STATE_WAITING_RIGHT = "WAITING_RIGHT";
  private static final int WIDTH = 90;
  private static final int HEIGHT = 170;
  private static final int RADIUS = 320;

  private JFrame frame = null;
  private JLabel lPicture = new JLabel();
  private JLabel lState = new JLabel();
  private String desire = DESIRE_THINK;
  private String state = STATE_THINKING;
  private int id = -1;
  private int forkIndexLeft = -1;
  private int forkIndexRight = -1;



  public AgentPhilosopher(int _id, int _forkIndexLeft, int
     _forkIndexRight)
  {
    id = _id;
    forkIndexLeft = _forkIndexLeft;
    forkIndexRight = _forkIndexRight;

    // Frame location
    int centreX = (int)(Toolkit.getDefaultToolkit().getScreenSize().
       getWidth() / 2.0);
    int centreY = (int)(Toolkit.getDefaultToolkit().getScreenSize().
       getHeight() / 2.0);
    double angle = (id) * 2.0 * Math.PI / 5.0 - Math.PI / 2.0;
    int offsetX = (int)(Math.cos(angle) * RADIUS);
    int offsetY = (int)(Math.sin(angle) * RADIUS);
    int x = centreX + offsetX - WIDTH / 2;
    int y = centreY + offsetY - HEIGHT / 2;

    // Setup GUI
    lPicture.setHorizontalAlignment(SwingConstants.CENTER);
    lPicture.setIcon(new ImageIcon("res/philosopher" + id + ".png"))
       ;
    lState.setHorizontalAlignment(SwingConstants.CENTER);
    lState.setOpaque(true);
    lState.setBorder(BorderFactory.createLoweredBevelBorder());
    JPanel contentPane = new JPanel(new BorderLayout());
    contentPane.add(lPicture, BorderLayout.CENTER);
    contentPane.add(lState, BorderLayout.SOUTH);
    frame = new JFrame(NAMES[id]);
    frame.setResizable(false);
    frame.setBounds(x, y, WIDTH, HEIGHT);
```

```
    frame.setContentPane(contentPane);
    frame.setVisible(true);

    updateGUI();
  }



  public boolean updateDesire() throws InterpreterException
  {
    if (state == STATE_WAITING_LEFT || state == STATE_WAITING_RIGHT)
      return true;

    int input = JOptionPane.showConfirmDialog(frame, "Do you want to
        eat?", frame.getTitle(), JOptionPane.YES_NO_OPTION);
    if (input == JOptionPane.YES_OPTION)
    {
      desire = DESIRE_EAT;

      if (state == STATE_THINKING)
        state = STATE_WAITING_LEFT;
    }
    else
      desire = DESIRE_THINK;

    return true;
  }



  public boolean wantsFork(Accessor ForkIndex) throws
     InterpreterException
  {
    boolean result = false;

    if (state == STATE_WAITING_LEFT)
    {
      ForkIndex.setValue(forkIndexLeft);
      result = true;
    }

    if (state == STATE_WAITING_RIGHT)
    {
      ForkIndex.setValue(forkIndexRight);
      result = true;
    }

    updateGUI();

    return result;
  }



  public boolean wantsStartThinking(Accessor ReturnedForkIndexLeft,
     Accessor ReturnedForkIndexRight) throws InterpreterException
  {
    boolean result = false;

    if (state == STATE_EATING && desire == DESIRE_THINK)
    {
      ReturnedForkIndexLeft.setValue(forkIndexLeft);
```

```
    ReturnedForkIndexRight.setValue(forkIndexRight);

    state = STATE_THINKING;
    result = true;
  }

  updateGUI();

  return result;
}



public boolean wantsNothing() throws InterpreterException
{
  boolean result = false;

  if (state == STATE_EATING && desire == DESIRE_EAT)
    result = true;

  if (state == STATE_THINKING && desire == DESIRE_THINK)
    result = true;

  updateGUI();

  return result;
}



public boolean gotFork(Accessor ForkIndex) throws
   InterpreterException
{
  if (ForkIndex.getValue().equals(forkIndexLeft))
    state = STATE_WAITING_RIGHT;

  else if (ForkIndex.getValue().equals(forkIndexRight))
    state = STATE_EATING;

  updateGUI();

  return true;
}



public boolean gotWait(Accessor ForkIndex) throws
   InterpreterException
{
  if (ForkIndex.getValue().equals(forkIndexLeft))
    state = STATE_WAITING_LEFT;

  else if (ForkIndex.getValue().equals(forkIndexRight))
    state = STATE_WAITING_RIGHT;

  updateGUI();

  return true;
}
```

```
  private void updateGUI()
  {
    if (state == STATE_WAITING_LEFT)
    {
      lState.setText("Needs left fork");
      lState.setBackground(Color.ORANGE);
    }
    else if (state == STATE_WAITING_RIGHT)
    {
      lState.setText("Needs right fork");
      lState.setBackground(Color.ORANGE);
    }
    else if (state == STATE_THINKING)
    {
      lState.setText("Is thinking...");
      lState.setBackground(Color.GREEN);
    }
    else if (state == STATE_EATING)
    {
      lState.setText("Is eating...");
      lState.setBackground(Color.RED);
    }
  }
}
```

Appendix B

LCC Protocols

B.1 Protocol *isolated*

```
r(solver, initial)
r(loop, cyclic)



a(solver, ID) ::

  a(loop, ID)



a(loop, ID) ::

  // Step the GA
  null <- __step()
```

B.2 Protocol *fitness*

```
r(solver, initial)
r(loop, cyclic)
r(requester(S), auxiliary)
r(receiverAll(S), auxiliary)
r(receiverOne(S), auxiliary)
r(responderSession(), uncommitted)



a(solver, ID) ::
  a(loop, ID)



a(loop, ID) ::
  null <- __step()
  then
  (
    (
```

```
      null <- __findPeers("", All) and __howManyToAsk(Ask) and
         __selectSomePeersAtRandom(All, Ask, Candidates)
      then
      null <- __chatter("# to ask:", Ask, "", "")
      then
      null <- __chatter("Asking the following random candidates:",
         Candidates, "", "")
      then
      a(requester(Candidates), ID)
      then
      a(receiverAll(Candidates), ID)
      then
      null <- __selectSession(Session)
      then
      null <- __breedWith(Session)
      then
      null <- __clearSessions()
    )
    or
    null <- __chatter("Alone", "", "", "")
  )



a(requester(Candidates), ID) ::
  null <- Candidates == []
  or
  (
    null <- Candidates = [H | Sr]
    then
    null <- __chatter("Asking candidate:", H, "", "")
    then
    askSession() => a(_, H)
    then
    a(requester(Sr), ID)
  )



a(receiverAll(Candidates), ID) ::

  null <- Candidates == []
  or
  (
    null <- Candidates = [Candidate | Sr]
    then
    a(receiverOne(Candidate), ID)
    then
    a(receiverAll(Sr), ID)
  )



a(receiverOne(Candidate), ID) ::

  // Respond to others' breeding requests
  a(responderSession(), ID)

  or

  // Receive response from candidate and breed
  (
```

```
    respondSession(CandidateSession) <= a(_, Candidate)
    then
    null <- __chatter("Considering session of:", Candidate, "", "")
    then
    null <- __considerSession(CandidateSession)
  )



a(responderSession(), ID) ::

  askSession() <= a(_, Requester)
  then
  null <- __chatter("Responding to:", Requester, "", "")
  then
  respondSession(MySession) => a(_, Requester) <- __getSession(
     MySession, Requester)
  then
  a(responderSession(), ID)
```

B.3 Protocol *memory*

```
r(solver, initial)
r(loop, cyclic)
r(listener(Selected), auxiliary)



a(solver, ID) ::

  // Kick-start (step once)
  null <- __step()

  then

  // Loop
  a(loop, ID)



a(loop, ID) ::

  // Store initial fitness
  null <- __getFitness(FitnessBefore)

  then

  // Find a mate and breed (respond to others while waiting for mate
     's response)
  (
    (
      null <- __findPeers("", Peers) and __selectOneMateFromMemory(
         Peers, Selected)
      then
      null <- __chatter("Mating with:", Selected, "", "")
      then
      request() => a(_, Selected)
      then
      a(listener(Selected), ID)
    )
```

```
    or
    null  // Alone
  )

  then

  // Step the GA
  null <- __step()

  then

  // Store final fitness
  null <- __getFitness(FitnessAfter)

  then

  // Update mate's ranking accordingly
  null <- __evaluateGain(FitnessBefore, FitnessAfter, Gain) and
      __updateMemory(Selected, Gain)



a(listener(Selected), ID) ::

  // Respond to others' requests
  (
    request() <= a(_, Requester)
    then
    respond(Session) => a(_, Requester) <- __getSession(Session,
        Requester)
    then
    a(listener(Selected), ID)
  )

  or

  // Receive response from mate and breed
  (
    respond(Session) <= a(_, Selected)
    then
    null <- __breedWith(Session)
  )
```

B.4 Protocol *central*

```
r(solver, initial)
r(loop, cyclic)
r(listener(Selected), auxiliary)



a(solver, ID) ::

  // Kick-start (step once)
  null <- __step()

  then

  // Loop
  a(loop, ID)
```

```
a(loop, ID) ::

  // Store initial fitness
  null <- __getFitness(FitnessBefore)

  then

  // Find a mate and breed (respond to others while waiting for mate
     's response)
  (
    (
      null <- __findPeers("", Peers) and __selectOneMateFromRankings
         (Peers, Selected)
      then
      null <- __chatter("Mating with:", Selected, "", "")
      then
      request() => a(_, Selected)
      then
      a(listener(Selected), ID)
    )
    or
    null  // Alone
  )

  then

  // Step the GA
  null <- __step()

  then

  // Store final fitness
  null <- __getFitness(FitnessAfter)

  then

  // Update mate's ranking accordingly
  null <- __evaluateGain(FitnessBefore, FitnessAfter, Gain) and
     __updateRanking(Selected, Gain)



a(listener(Selected), ID) ::

  // Respond to others' requests
  (
    request() <= a(_, Requester)
    then
    respond(Session) => a(_, Requester) <- __getSession(Session,
       Requester)
    then
    a(listener(Selected), ID)
  )

  or

  // Receive response from mate and breed
  (
    respond(Session) <= a(_, Selected)
```

```
    then
    null <- __breedWith(Session)
  )
```

B.5 Protocol *collective*

```
r(solver, initial)
r(loop, cyclic)
r(askerOfAllSuggestions(Suggesters), auxiliary)
r(listenerOfAllSuggestions(Suggesters), auxiliary)
r(responderSuggestion(), uncommitted)
r(responderSession(), uncommitted)



a(solver, ID) ::

  // Kick-start (step once)
  null <- __step()

  then

  // Loop
  a(loop, ID)



a(loop, ID) ::

  // Store initial fitness
  null <- __getFitness(FitnessBefore)

  then

  // Find suggesters
  null <- __findPeers("", Pool) and __howManyToAsk(Ask) and
     __selectSomeSuggestersFromHistory(Pool, Ask, SelectedList)
  then
  null <- __chatter("Selected suggesters: ", SelectedList, "", "")
  then
  null <- __initSuggestionTable(Pool)
  then
  a(askerOfAllSuggestions(SelectedList), ID)
  then
  a(listenerOfAllSuggestions(SelectedList), ID)

  then

  // Breed (respond to others while waiting for peer's response)
  null <- __selectOneMateFromSuggestions(Mate)
  then
  null <- __chatter("Hitting on: ", Mate, "", "")
  then
  requestSession() => a(_, Mate)
  then
  (
    // Respond to others' suggestion requests
    a(responderSuggestion(), ID)

    or
```

```
    // Respond to others' breeding requests
    a(responderSession(), ID)

    or

    // Receive response from mate and breed
    (
      respondSession(TheirSession) <= a(_, Mate)
      then
      null <- __chatter("Mating with:", Mate, "", "")
      then
      null <- __breedWith(TheirSession)
    )
  )

  then

  // Step the GA
  null <- __step()

  then

  // Store final fitness
  null <- __getFitness(FitnessAfter)

  then

  // Update mate's ranking accordingly
  null <- __evaluateGain(FitnessBefore, FitnessAfter, Gain) and
      __updateHistory(Mate, Gain)
  then
  null <- __chatter(Mate, " gained me ", Gain, "")



a(askerOfAllSuggestions(Suggesters), ID) ::

  null <- Suggesters == []
  or
  (
    null <- Suggesters = [H | Sr]
    then
    null <- __chatter("Requesting suggestions from: ", H, "", "")
    then
    requestSuggestion() => a(_, H)
    then
    a(askerOfAllSuggestions(Sr), ID)
  )



a(listenerOfAllSuggestions(Suggesters), ID) ::

  null <- Suggesters == []
  or
  (
    null <- Suggesters = [H | Sr]
    then
    (
      // Respond to others' suggestion requests
      a(responderSuggestion(), ID)
```

```
      or

      // Respond to others' breeding requests
      a(responderSession(), ID)

      or

      // Receive suggestion
      (
        respondSuggestions(TheirHistory) <= a(_, Suggester)
        then
        //null <- __chatter("Got history of: ", Suggester, "", "")
        //then
        null <- __augmentSuggestions(TheirHistory)
      )

      or

      // Receive nothing
      (
        respondNothing() <= a(_, Suggester)
        then
        null <- __chatter("Got no suggestions from ", Suggester, "",
            "")
      )
    )
    then
    a(listenerOfAllSuggestions(Sr), ID)
  )



a(responderSuggestion(), ID) ::

  requestSuggestion() <= a(_, Requester)
  then
  (
    (
      respondSuggestions(MyHistory) => a(_, Requester)  <-
         __getHistory(Requester, MyHistory)
      then
      null <- __chatter("Sent my history to: ", Requester, "", "")
    )
    or
    (
      respondNothing() => a(_, Requester)
      then
      null <- __chatter("Nothing to suggest to ", Requester, "", "")
    )
  )
  then
  a(responderSuggestion(), ID)



a(responderSession(), ID) ::

  requestSession() <= a(_, Requester)
  then
  respondSession(Session) => a(_, Requester) <- __getSession(Session
     , Requester)
```

```
then
a(responderSession(), ID)
```

Appendix C

Java Source Code

C.1 Class *Main*

```
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Arrays;
import java.util.List;


import lij.runtime.Interpreter;



public class Main
{
  public static void main(String[] args) throws Exception
  {
    // Print the arguments
    System.out.println(Arrays.toString(args));

    // Parse options
    Options options = new Options(args);

    // Initialise interpreter
    String protocolName = options.getString(Constants.
       OPTION_PROTOCOL);
    InputStream is = new FileInputStream(protocolName);
    Interpreter interpreter = new Interpreter(is, options.getBoolean
       (Constants.OPTION_GUI) && options.getBoolean(Constants.
       OPTION_MONITOR));

    // Determine damaged ids
    int n = options.getInteger(Constants.OPTION_N);
    int nDamaged = options.getInteger(Constants.OPTION_DMG);
    List<Integer> idDamaged = Arrays.asList(Utilities.chooseRandom
       (0, n, nDamaged));

    // Subscribe solvers
    for (int i = 0; i < options.getInteger(Constants.OPTION_N); i++)
      interpreter.subscribe("solver", new AgentSolver(i, options,
         idDamaged.contains(i)), i);
```

```
    // Start the interpreter
    interpreter.run();
  }
}
```

C.2 Class *AgentSolver*

```
import java.io.Serializable;
import java.lang.reflect.Constructor;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;

import jeva.ga.Breeder;
import jeva.ga.Evaluator;
import jeva.ga.Objective;
import jeva.ga.Parameters;
import jeva.ga.Population;
import jeva.ga.Selectable;
import jeva.ga.Selector;
import jeva.ga.objective.ObjectiveMaximize;
import jeva.ga.objective.ObjectiveMinimize;
import jeva.ga.selector.SelectorBest;
import jeva.ga.selector.SelectorRouletteRebased;
import jeva.ga.selector.SelectorTournament;
import lij.exceptions.InterpreterException;
import lij.interfaces.Accessor;
import lij.interfaces.ConstraintImplementor;



public class AgentSolver implements ConstraintImplementor
{
  private static int agentIndex = 0;
  private static int nFinished = 0;

  private Serializable myId = null;
  private String name = "";
  private AgentSolverFrame frame = null;
  private Options options = null;
  private Breeder breeder = null;
  private Parameters parameters = new Parameters();
  private Objective objective = null;
  private Objective objectiveGain = new ObjectiveMaximize();
  private boolean meFinished = false;

  private ArrayList<Session> sessions = new ArrayList<Session>();
  private HashMap<Serializable, Double> history = new HashMap<
      Serializable, Double>();
  private HashMap<Serializable, Double> memory = new HashMap<
      Serializable, Double>();
  private HashMap<Serializable, Double> suggestions = new HashMap<
      Serializable, Double>();
  private static HashMap<Serializable, Double> rankings = new
      HashMap<Serializable, Double>();
  private HashMap<Serializable, Integer> countsConsulted = new
      HashMap<Serializable, Integer>();
```

```
private HashMap<Serializable, Integer> countsConsultedBy = new
    HashMap<Serializable, Integer>();
private HashMap<Serializable, Integer> countsSexed = new HashMap<
    Serializable, Integer>();
private HashMap<Serializable, Integer> countsSexedBy = new HashMap
    <Serializable, Integer>();
private HashMap<Serializable, Integer> countsSuggested = new
    HashMap<Serializable, Integer>();
private int totalSexedBy = 0;
private int totalConsultedBy = 0;
private int lastSex = 0;

private boolean meDamaged = false;

// This is cheating, but is only used for rendering the history
    table in the GUI (damaged agents = red rows)
protected static ArrayList<Serializable> listOfDamagedAgentsCheat
    = new ArrayList<Serializable>();



public AgentSolver(Serializable _myId, Options _options, boolean
    _meDamaged)
{
  myId = _myId;
  options = _options;
  meDamaged = _meDamaged;

  name = (meDamaged ? "SOLVER_" : "Solver_") + myId; // CAPS name
      identify damaged agent

  // This is cheating, but is only used for rendering the history
      table in the GUI (damaged agents = red rows)
  if (meDamaged)
  {
    log("damaged");
    listOfDamagedAgentsCheat.add(myId);
  }

  // Initialise breeder parameters
  parameters.put(Parameters.GENOME_LENGTH, options.getInteger(
      Constants.OPTION_NVARS) * options.getInteger(Constants.
      OPTION_NBITS));
  parameters.put(Parameters.POPULATION_SIZE, options.getInteger(
      Constants.OPTION_PS));
  parameters.put(Parameters.ELITE_SIZE, options.getInteger(
      Constants.OPTION_EP));
  parseDynamicParameter(Parameters.CROSSOVER_RATE, options.
      getString(Constants.OPTION_CR));
  parseDynamicParameter(Parameters.MUTATION_RATE, options.
      getString(Constants.OPTION_MR));
  parameters.put(Parameters.INITIALIZER, Utilities.createObject(
      options.getString(Constants.OPTION_IF)));
  parameters.put(Parameters.SELECTOR, Utilities.createObject(
      options.getString(Constants.OPTION_SF)));
  parameters.put(Parameters.CROSSOVERER, Utilities.createObject(
      options.getString(Constants.OPTION_CF)));
  parameters.put(Parameters.MUTATOR, Utilities.createObject(
      options.getString(Constants.OPTION_MF)));

  // Initialise evaluator
  Evaluator evaluator = null;
```

```
  try
  {
    Class<?> theClass = (Class<?>)Class.forName(options.getString(
        Constants.OPTION_EF));
    Constructor<?> constructor = (Constructor<?>)theClass.
        getConstructor(new Class[] { int.class, int.class });
    evaluator = (Evaluator)(constructor.newInstance(options.
        getInteger(Constants.OPTION_NVARS), options.getInteger(
        Constants.OPTION_NBITS)));
  }
  catch (Exception e)
  {
    e.printStackTrace();
    System.exit(-1);
  }

  // Initialise objective
  objective = (options.getString(Constants.OPTION_OBJ).equals("min
      ") ? new ObjectiveMinimize() : new ObjectiveMaximize());

  // Initialise breeder
  breeder = new Breeder(evaluator, objective, parameters,
      Constants.N_EVALUATION_THREADS, Constants.HISTORY_LENGTH);

  // Show frame
  if (options.getBoolean(Constants.OPTION_GUI))
    frame = new AgentSolverFrame(name, this, options.getDouble(
        Constants.OPTION_TFIT), options.getDouble(Constants.
        OPTION_CYCLE), meDamaged);

  agentIndex++;
}



private void parseDynamicParameter(String key, String value)
{
  if (value.equals("rnd"))
    parameters.put(key, Math.random());
  else if (value.equals("half"))
    parameters.put(key, 1 / Math.pow(2, agentIndex));
  else if (value.equals("div"))
    parameters.put(key, 1 / Math.pow(10, agentIndex));
  else if (Utilities.isArrayDouble(value))
  {
    Double[] cr = Utilities.parseArrayDouble(value);
    parameters.put(key, cr[agentIndex % cr.length]);
  }
  else if (Utilities.isDouble(value))
    parameters.put(key, Double.parseDouble(value));
  else
  {
    System.out.println("ERROR: Bad value \"" + value + "\"
        specified for parameter: " + key);
    System.exit(1);
  }
}



private double breedParameter(double parameter1, double parameter2
    )
```

```
{
  double newParameter = (parameter1 + parameter2) / 2.0;
  newParameter += (Math.random() * 2 - 1) * newParameter;
  newParameter = Utilities.limit(newParameter, Constants.MIN_MR,
     1);
  return newParameter;
}



private void log(String s)
{
  if (frame != null)
    frame.log(s);

  System.out.println(name + ">\t" + s);
}



public Breeder getBreeder()
{
  return breeder;
}



public int getTotalSexedBy()
{
  return totalSexedBy;
}



public int getTotalConsultedBy()
{
  return totalConsultedBy;
}



public int getLastSex()
{
  return lastSex;
}



public double getHistoryFor(Serializable id)
{
  return history.containsKey(id) ? history.get(id) : 0.0; // 1.0
     // ???
}



public double getMemoryFor(Serializable id)
{
  return memory.containsKey(id) ? memory.get(id) : 0.0; // 1.0  //
      ???
}
```

```
public double getSuggestionsFor(Serializable id)
{
  return suggestions.containsKey(id) ? suggestions.get(id) : 0.0;
     // 1.0  // ???
}



public synchronized double getRankingFor(Serializable id)
{
  return rankings.containsKey(id) ? rankings.get(id) : 0.0; // 1.0
       // ???
}



public int getCountConsultedFor(Serializable id)
{
  return countsConsulted.containsKey(id) ? countsConsulted.get(id)
      : 0;
}



public int getCountConsultedByFor(Serializable id)
{
  return countsConsultedBy.containsKey(id) ? countsConsultedBy.get
     (id) : 0;
}



public int getCountSexedFor(Serializable id)
{
  return countsSexed.containsKey(id) ? countsSexed.get(id) : 0;
}



public int getCountSexedByFor(Serializable id)
{
  return countsSexedBy.containsKey(id) ? countsSexedBy.get(id) :
     0;
}



public int getCountSuggestedFor(Serializable id)
{
  return countsSuggested.containsKey(id) ? countsSuggested.get(id)
      : 0;
}



public ArrayList<Serializable> listAllKnownPeers()
{
  HashSet<Serializable> union = new HashSet<Serializable>();
  union.addAll(history.keySet());
  union.addAll(memory.keySet());
```

```
  union.addAll(suggestions.keySet());
  //    union.addAll(rankings.keySet());  // Don't use this -
     otherwise self will appear in table!
  union.addAll(countsConsulted.keySet());
  union.addAll(countsConsultedBy.keySet());
  union.addAll(countsSexed.keySet());
  union.addAll(countsSexedBy.keySet());
  union.addAll(countsSuggested.keySet());
  return new ArrayList<Serializable>(union);
}



////////////////////////////////////////// LCC constraint
    methods start here

// LCC
public boolean __chatter(Accessor __text1, Accessor __text2,
    Accessor __text3, Accessor __text4)
{
  if (options.getBoolean(Constants.OPTION_CHATTER))
    log(__text1.getValue().toString() + __text2.getValue().
        toString() + __text3.getValue().toString() + __text4.
        getValue().toString());

  return true;
}



// LCC
public boolean __howManyToAsk(Accessor __ask) throws
    InterpreterException
{
  int nAsk = options.getInteger(Constants.OPTION_ASK);

  __ask.setValue(nAsk);

  return true;
}



// LCC
public boolean __evaluateGain(Accessor __fitnessBefore, Accessor
    __fitnessAfter, Accessor __gain) throws InterpreterException
{
  double fitnessBeforeValue = (Double)(__fitnessBefore.getValue())
      ;
  double fitnessAfterValue = (Double)(__fitnessAfter.getValue());

  double gain = fitnessAfterValue - fitnessBeforeValue;
  if (objective instanceof ObjectiveMinimize)
    gain *= -1;

  __gain.setValue(gain);

  return true;
}
```

```
// LCC [fitness, random(1)]
public boolean __selectSomePeersAtRandom(Accessor __pool, Accessor
    __n, Accessor __selectedList) throws InterpreterException
{
  @SuppressWarnings("unchecked")
  ArrayList<Serializable> poolIDs = (ArrayList<Serializable>)((
     ArrayList<Serializable>)__pool.getValue()).clone();
  ArrayList<Serializable> resultIDs = new ArrayList<Serializable
     >();
  int n = (Integer)(__n.getValue());

  // Assert
  if (poolIDs.size() < n)
  {
    log("ERROR: Not enough peers in pool for selecting at random (
       available: " + poolIDs.size() + " - requested: " + n + ")")
       ;
    return false;
  }

  // Select
  for (int i = 0; i < n; i++)
  {
    int idx = (int)(Math.random() * poolIDs.size());
    resultIDs.add(poolIDs.remove(idx));
  }

  __selectedList.setValue(resultIDs);

  return true;
}



// LCC [collective]
public boolean __selectSomeSuggestersFromHistory(Accessor __pool,
    Accessor __n, Accessor __selectedList) throws
    InterpreterException
{
  @SuppressWarnings("unchecked")
  ArrayList<Serializable> poolIDs = (ArrayList<Serializable>)((
     ArrayList<Serializable>)__pool.getValue()).clone();
  ArrayList<Serializable> resultIDs = new ArrayList<Serializable
     >();
  int n = (Integer)(__n.getValue());

  // Assert
  if (poolIDs.size() < n)
  {
    log("ERROR: Not enough peers in pool for selecting suggesters
       (available: " + poolIDs.size() + " - requested: " + n + ")"
       );
    return false;
  }

  // Create selectables
  Selectable[] selectables = new Selectable[poolIDs.size()];
  for (int i = 0; i < selectables.length; i++)
    selectables[i] = new SelectableAgentWrapper(poolIDs.get(i),
       getHistoryFor(poolIDs.get(i)));
  Arrays.sort(selectables, objectiveGain);
```

```
  // Select
  Selector selector = new SelectorBest();
  //Selector selector = new SelectorRouletteRebased();
  Selectable[] selected = selector.select(objectiveGain,
      selectables, n);
  for (int i = 0; i < n; i++)
  {
    Serializable id = ((SelectableAgentWrapper)selected[i]).
        getAgentID();
    resultIDs.add(id);
    countsConsulted.put(id, getCountConsultedFor(id) + 1);
  }

  __selectedList.setValue(resultIDs);

  return true;
}



// LCC [collective]
public boolean __selectOneMateFromSuggestions(Accessor __result)
    throws InterpreterException
{
  // Assert
  if (suggestions.size() <= 0)
  {
    log("ERROR: Suggestions table is empty");
    return false;
  }

  // Create selectables
  Serializable[] ids = suggestions.keySet().toArray(new
      Serializable[0]);
  Selectable[] selectables = new Selectable[ids.length];
  for (int i = 0; i < selectables.length; i++)
    selectables[i] = new SelectableAgentWrapper(ids[i],
        suggestions.get(ids[i]));
  Arrays.sort(selectables, objectiveGain);

  // Select
  //Selector selector = new SelectorBest();
  //Selector selector = new SelectorRouletteRebased();
  int k;
  k = options.getInteger(Constants.OPTION_DMG) * 2;
  k = Utilities.limit(k, 2, options.getInteger(Constants.OPTION_N)
      - 1);
  Selector selector = new SelectorTournament(k);
  SelectableAgentWrapper selected = (SelectableAgentWrapper)(
      selector.select(objectiveGain, selectables, 1)[0]);

  __result.setValue(selected.getAgentID());

  return true;
}



// LCC [memory]
public boolean __selectOneMateFromMemory(Accessor __pool, Accessor
    __selected) throws InterpreterException
{
```

```
  @SuppressWarnings("unchecked")
  ArrayList<Serializable> poolIDs = (ArrayList<Serializable>)((
      ArrayList<Serializable>)__pool.getValue()).clone();

  // Assert
  if (poolIDs.size() < 1)
  {
    log("ERROR: Not enough peers in pool for selecting a mate from
        memory");
    return false;
  }

  // Create selectables
  Selectable[] selectables = new Selectable[poolIDs.size()];
  for (int i = 0; i < selectables.length; i++)
    selectables[i] = new SelectableAgentWrapper(poolIDs.get(i),
       getMemoryFor(poolIDs.get(i)));
  Arrays.sort(selectables, objectiveGain);

  // Select
  //Selector selector = new SelectorBest();
  //Selector selector = new SelectorRouletteRebased();
  int k;
  k = options.getInteger(Constants.OPTION_DMG) * 2;
  k = Utilities.limit(k, 2, options.getInteger(Constants.OPTION_N)
      - 1);
  Selector selector = new SelectorTournament(k);
  Selectable selected = selector.select(objectiveGain, selectables
      , 1)[0];

  __selected.setValue(((SelectableAgentWrapper)selected).
      getAgentID());

  return true;
}



// LCC [central]
public boolean __selectOneMateFromRankings(Accessor __pool,
    Accessor __selected) throws InterpreterException
{
  @SuppressWarnings("unchecked")
  ArrayList<Serializable> poolIDs = (ArrayList<Serializable>)((
      ArrayList<Serializable>)__pool.getValue()).clone();

  // Assert
  if (poolIDs.size() < 1)
  {
    log("ERROR: Not enough peers in pool for selecting a mate from
        rankings");
    return false;
  }

  // Create selectables
  Selectable[] selectables = new Selectable[poolIDs.size()];
  for (int i = 0; i < selectables.length; i++)
    selectables[i] = new SelectableAgentWrapper(poolIDs.get(i),
       getRankingFor(poolIDs.get(i)));
  Arrays.sort(selectables, objectiveGain);

  // Select
```

```
  //Selector selector = new SelectorBest();
  //Selector selector = new SelectorRouletteRebased();
  int k;
  k = options.getInteger(Constants.OPTION_DMG) * 2;
  k = Utilities.limit(k, 2, options.getInteger(Constants.OPTION_N)
      - 1);
  Selector selector = new SelectorTournament(k);
  Selectable selected = selector.select(objectiveGain, selectables
     , 1)[0];

  __selected.setValue(((SelectableAgentWrapper)selected).
     getAgentID());

  return true;
}



// LCC [collective]
public boolean __updateHistory(Accessor __id, Accessor __gain)
    throws InterpreterException
{
  Serializable key = __id.getValue();
  Double newGain = (Double)(__gain.getValue());

  if (options.getBoolean(Constants.OPTION_CUMULATIVE))
    newGain += getHistoryFor(key);

  history.put(key, newGain);

  countsSexed.put(key, getCountSexedFor(key) + 1);

  if (frame != null)
    frame.selectHistoryRow(key);

  return true;
}



// LCC [memory]
public boolean __updateMemory(Accessor __id, Accessor __gain)
    throws InterpreterException
{
  Serializable key = __id.getValue();
  Double newGain = (Double)(__gain.getValue());

  newGain += getMemoryFor(key);
  memory.put(key, newGain);

  countsSexed.put(key, getCountSexedFor(key) + 1);

  if (frame != null)
    frame.selectHistoryRow(key);

  return true;
}



// LCC [central]
public synchronized boolean __updateRanking(Accessor __id,
```

```
    Accessor __gain) throws InterpreterException
{
  Serializable key = __id.getValue();
  //double score = (Double)(__gain.getValue()) > 0 ? 1.0 : -1.0;
  double score = (Double)(__gain.getValue());

  score += getRankingFor(key);
  rankings.put(key, score);

  countsSexed.put(key, getCountSexedFor(key) + 1);

  if (frame != null)
    frame.selectHistoryRow(key);

  return true;
}



// LCC [collective]
public boolean __getHistory(Accessor __requester, Accessor
    __myHistory) throws InterpreterException
{
  @SuppressWarnings("unchecked")
  HashMap<Serializable, Double> historyCopy = (HashMap<
      Serializable, Double>)(((HashMap<Serializable, Double>)
      history).clone());

  Serializable requesterKey = __requester.getValue();
  countsConsultedBy.put(requesterKey, getCountConsultedByFor(
      requesterKey) + 1);
  totalConsultedBy++;

  __myHistory.setValue(historyCopy);

  return true;
}



// LCC [collective]
public boolean __augmentSuggestions(Accessor __theirHistory)
    throws InterpreterException
{
  @SuppressWarnings("unchecked")
  HashMap<Serializable, Double> theirHistory = (HashMap<
      Serializable, Double>)__theirHistory.getValue(); // Cloned by
       provider

  for (Serializable key : theirHistory.keySet())
  {
    if (key.equals(myId)) // Skip self
      continue;

    double myValue = getSuggestionsFor(key);
    double theirValue = theirHistory.get(key);

    //      if (options.getBoolean(Constants.OPTION_AVERAGE))
    //        theirValue /= (options.getInteger(Constants.
        OPTION_ASK) + 1); // +1 for self

    suggestions.put(key, (myValue + theirValue));
```

```
    }

    return true;
  }



  // LCC [collective]
  public boolean __initSuggestionTable(Accessor __pool) throws
      InterpreterException
  {
    suggestions.clear();

    @SuppressWarnings("unchecked")
    ArrayList<Serializable> ids = (ArrayList<Serializable>)__pool.
        getValue();

    for (Serializable id : ids)
    {
      double value = getHistoryFor(id);

      //      if (options.getBoolean(Constants.OPTION_AVERAGE))
      //        value /= (options.getInteger(Constants.OPTION_ASK) +
          1); // +1 for self

      suggestions.put(id, value);
    }

    return true;
  }



  // LCC [fitness]
  public boolean __considerSession(Accessor __session)
  {
    Session session = (Session)(__session.getValue());
    sessions.add(session);

    return true;
  }



  // LCC [fitness]
  public boolean __selectSession(Accessor __session) throws
      InterpreterException
  {
    // Create selectables
    Collections.sort(sessions, objective);
    Selectable[] selectables = sessions.toArray(new Selectable[] {})
        ;

    // Select
    Selector selector = new SelectorRouletteRebased();
    //Selector selector = new SelectorBest();
    Session selected = (Session)(selector.select(objective,
        selectables, 1)[0]);

    __session.setValue(selected);

    return true;
```

```
}


// LCC [fitness]
public boolean __clearSessions()
{
  sessions.clear();

  return true;
}


// LCC
public boolean __getFitness(Accessor __fitness) throws
    InterpreterException
{
  if (breeder.getGeneration() == 0)
    __fitness.setValue(0); // ???
  else
    __fitness.setValue(breeder.getLastPopulation().getFitnessMean
        ());

  return true;
}


// LCC
public boolean __getSession(Accessor __session, Accessor
    __requester) throws InterpreterException
{
  Serializable key = __requester.getValue();
  countsSexedBy.put(key, getCountSexedByFor(key) + 1);
  totalSexedBy++;
  lastSex = breeder.getGeneration();

  Population myPopulation = (Population)(breeder.getLastPopulation
      ().clone());
  Parameters myParameters = (Parameters)(breeder.getParameters().
      clone());
  Session session = new Session(myPopulation, myParameters,
      meDamaged);

  __session.setValue(session);

  return true;
}


// LCC
public boolean __breedWith(Accessor __session)
{
  if (breeder.getGeneration() == 0)
    return true;

  if (meDamaged)
    return true;

  Session session2 = (Session)(__session.getValue());
```

```
  // Population
  if (options.getString(Constants.OPTION_TYPE).equals("pop") ||
      options.getString(Constants.OPTION_TYPE).equals("full"))
  {
    Population thisPopulation = breeder.getLastPopulation();
    Population otherPopulation = session2.getPopulation();
    if (thisPopulation == null || otherPopulation == null)
      return true;
    for (int i = 0; i < thisPopulation.getSize(); i++)
      if (Math.random() >= 0.5)
        thisPopulation.setGenome(i, otherPopulation.getGenome(i));
    thisPopulation.sort();
  }

  // Parameters
  if (options.getString(Constants.OPTION_TYPE).equals("par") ||
      options.getString(Constants.OPTION_TYPE).equals("full"))
  {
    // mr
    double newMr = breedParameter(breeder.getParameters().
        getDouble(Parameters.MUTATION_RATE), session2.getParameters
        ().getDouble(Parameters.MUTATION_RATE));
    breeder.getParameters().put(Parameters.MUTATION_RATE, newMr);

    //      // cr
    //      double newCr = breedParameter(breeder.getParameters().
        getDouble(Parameters.CROSSOVER_RATE), session2.
        getParameters().getDouble(Parameters.CROSSOVER_RATE));
    //      breeder.getParameters().put(Parameters.CROSSOVER_RATE,
         newCr);
  }

  return true;
}



// LCC
public boolean __step()
{
  boolean finished = false;

  for (int i = 0; i < options.getInteger(Constants.OPTION_CYCLE)
      && !finished; i++)
  {
    // Step the GA
    if (meDamaged)
    {
      if (breeder.getGeneration() == 0) // Kickstart breeder
      {
        breeder.step();
        breeder.getLastPopulation().damage();
        breeder.getBestPopulation().damage();
        breeder.getParameters().damage();
      }
      else
        breeder.skip();
    }
    else
      breeder.step();
```

```
      // Check stop conditions
      if (options.getDouble(Constants.OPTION_TFIT) != null &&
          breeder.getBestPopulation().getFitnessBest() <= options.
          getDouble(Constants.OPTION_TFIT))
      {
        log("Reached required fitness: " + options.getDouble(
            Constants.OPTION_TFIT) + " in " + breeder.getGeneration()
             + " generations.");
        if (!options.getBoolean(Constants.OPTION_GUI))
          System.exit(0);
        finished = true;
      }
      if (options.getInteger(Constants.OPTION_TGEN) != null &&
          breeder.getGeneration() >= options.getInteger(Constants.
          OPTION_TGEN))
      {
        if (!meFinished)
        {
          log("Reached required number of generations: " + options.
              getInteger(Constants.OPTION_TGEN) + " with a best
              fitness of " + breeder.getBestPopulation().
              getFitnessBest());
          nFinished++;
          meFinished = true;
        }
        if (nFinished == agentIndex) // Last one
        {
          if (!options.getBoolean(Constants.OPTION_GUI))
            System.exit(0);
          finished = true;
        }
      }

      // Yield
      Utilities.sleep(1);
    }

    // Update info in GUI
    if (frame != null)
      frame.updateGUI();

    // Display info
    if (options.getBoolean(Constants.OPTION_PROGRESS) && !finished)
      log("g = " + breeder.getGeneration() + "\tsexed = " +
          totalSexedBy + "\tf = " + breeder.getBestPopulation().
          getFitnessMean() + "\tmr = " + breeder.getParameters().
          getDouble(Parameters.MUTATION_RATE));

    return !finished;
  }
}
```

C.3 Class *Session*

```
import java.io.Serializable;

import jeva.ga.Parameters;
import jeva.ga.Population;
import jeva.ga.Selectable;
```

```
public class Session implements Selectable, Serializable
{
  private Population population;
  private Parameters parameters;
  private boolean damaged = false;


  public Session(Population _population, Parameters _parameters,
     boolean _damaged)
  {
    population = _population;
    parameters = _parameters;
    damaged = _damaged;
  }


  public Population getPopulation()
  {
    return population;
  }


  public Parameters getParameters()
  {
    return parameters;
  }


  public double getFitness()
  {
    return damaged ? -1 * Double.MAX_VALUE : population.
       getFitnessMean(); // was: -100000.0
  }


  public boolean isDamaged()
  {
    return damaged;
  }


  public String toString()
  {
    return String.valueOf(getFitness());
  }
}
```

C.4 Class *SelectableAgentWrapper*

```
import java.io.Serializable;
```

```
import jeva.ga.Selectable;



public class SelectableAgentWrapper implements Selectable
{
  private Serializable agentID;
  private double fitness;



  public SelectableAgentWrapper(Serializable _agentID, double
     _fitness)
  {
    agentID = _agentID;
    fitness = _fitness;
  }



  public Serializable getAgentID()
  {
    return agentID;
  }



  public double getFitness()
  {
    return fitness;
  }



  public String toString()
  {
    return agentID + " = " + getFitness();
  }

}
```

C.5 Class *AgentSolverFrame*

```
import java.awt.BorderLayout;
import java.awt.Toolkit;
import java.io.Serializable;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTabbedPane;
import javax.swing.JTable;

import lij.monitor.LogArea;
```

```
public class AgentSolverFrame extends JFrame
{
  public static final int SCREEN_WIDTH = (int)Toolkit.
     getDefaultToolkit().getScreenSize().getWidth();
  public static final int SCREEN_HEIGHT = (int)Toolkit.
     getDefaultToolkit().getScreenSize().getHeight();
  public static final int FRAME_WIDTH = SCREEN_WIDTH / 4;
  public static final int FRAME_HEIGHT = (SCREEN_HEIGHT - 24) / 2;
  public static final int INFO_SPLIT_PANE_DIVIDER_LOCATION = (int)
     (1.0 / 2.0 * (double)FRAME_WIDTH);

  private static int lastX = 0;
  private static int lastY = 0;
  private AgentSolver solver = null;
  private LogArea logArea = new LogArea();
  private GraphFitness graphFitness;
  private TableModelStatistics tableModelStatistics;
  private TableModelHistory tableModelHistory;
  private TableModelCounts tableModelCounts;
  private JTable tableInfo;
  private JTable tableHistory;
  private JTable tableCounts;



  public AgentSolverFrame(String title, AgentSolver _solver, Double
     _tfit, Double _cycle, boolean meDamaged)
  {
    super(title);

    solver = _solver;

    // Setup components
    graphFitness = new GraphFitness(solver.getBreeder(), _tfit,
       _cycle, meDamaged);

    tableModelStatistics = new TableModelStatistics(solver);
    tableInfo = new JTable(tableModelStatistics);

    tableModelHistory = new TableModelHistory(solver);
    tableHistory = new JTable(tableModelHistory);
    tableHistory.setDefaultRenderer(Object.class, new
       TableCellRendererHistory(solver));
    int tableHistoryColumnIndex = 0;
    tableHistory.getColumnModel().getColumn(tableHistoryColumnIndex
       ++).setHeaderValue("ID");
    tableHistory.getColumnModel().getColumn(tableHistoryColumnIndex
       ++).setHeaderValue("History");
    tableHistory.getColumnModel().getColumn(tableHistoryColumnIndex
       ++).setHeaderValue("Suggestions");
    tableHistory.getColumnModel().getColumn(tableHistoryColumnIndex
       ++).setHeaderValue("Memory");
    tableHistory.getColumnModel().getColumn(tableHistoryColumnIndex
       ++).setHeaderValue("Rankings");

    tableModelCounts = new TableModelCounts(solver);
    tableCounts = new JTable(tableModelCounts);
    tableCounts.setDefaultRenderer(Object.class, new
       TableCellRendererHistory(solver));
    int tableCountsColumnIndex = 0;
```

```
    tableCounts.getColumnModel().getColumn(tableCountsColumnIndex++)
        .setHeaderValue("ID");
    tableCounts.getColumnModel().getColumn(tableCountsColumnIndex++)
        .setHeaderValue("Consulted");
    tableCounts.getColumnModel().getColumn(tableCountsColumnIndex++)
        .setHeaderValue("ConsultedBy");
    tableCounts.getColumnModel().getColumn(tableCountsColumnIndex++)
        .setHeaderValue("Sexed");
    tableCounts.getColumnModel().getColumn(tableCountsColumnIndex++)
        .setHeaderValue("SexedBy");
    tableCounts.getColumnModel().getColumn(tableCountsColumnIndex++)
        .setHeaderValue("Suggested");

    logArea.setShowTimestamp(false);

    // Setup window
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setBounds(lastX, lastY, FRAME_WIDTH, FRAME_HEIGHT);
    lastX += FRAME_WIDTH;
    if (lastX + FRAME_WIDTH > SCREEN_WIDTH)
    {
      lastX = 0;
      lastY += FRAME_HEIGHT;
    }

    // Layout window
    JSplitPane spInfo = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
    spInfo.setLeftComponent(graphFitness);
    spInfo.setRightComponent(tableInfo);
    spInfo.setContinuousLayout(true);
    spInfo.setDividerLocation(INFO_SPLIT_PANE_DIVIDER_LOCATION);
    JTabbedPane tpLog = new JTabbedPane();
    tpLog.addTab("Log", logArea);
    tpLog.addTab("History", new JScrollPane(tableHistory));
    tpLog.addTab("Counts", new JScrollPane(tableCounts));
    tpLog.setSelectedIndex(1);
    JSplitPane spLog = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
    spLog.setLeftComponent(spInfo);
    spLog.setRightComponent(tpLog);
    spLog.setContinuousLayout(true);
    spLog.setOneTouchExpandable(true);
    getContentPane().add(spLog, BorderLayout.CENTER);

    // Show
    setVisible(true);
    if (meDamaged)
      setState(JFrame.ICONIFIED);
  }



  public void log(String s)
  {
    logArea.append(s);
  }



  public void selectHistoryRow(Serializable key)
  {
    for (int i = 0; i < tableModelHistory.getRowCount(); i++)
      if (tableModelHistory.getValueAt(i, 0) == key)
```

```
      {
        tableHistory.getSelectionModel().setSelectionInterval(i, i);
        break;
      }

    for (int i = 0; i < tableModelCounts.getRowCount(); i++)
      if (tableModelCounts.getValueAt(i, 0) == key)
      {
        tableCounts.getSelectionModel().setSelectionInterval(i, i);
        break;
      }
  }



  public void updateGUI()
  {
    tableModelStatistics.fireTableDataChanged();
    tableModelHistory.fireTableDataChanged();
    tableModelCounts.fireTableDataChanged();
    graphFitness.repaint();
  }
}
```

C.6 Class *LogArea*

```
import java.awt.BorderLayout;
import java.awt.Toolkit;
import java.awt.datatransfer.StringSelection;
import java.util.ArrayList;
import java.util.Date;

import javax.swing.JComponent;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.text.BadLocationException;



/**
 * This class defines a component containing a JTextArea, and
    provides
 * convenience methods for logging text into it.
 *
 * @author Nikolaos Chatzinikolaou
 * @version 2005.12.16
 */
public class LogArea extends JComponent
{
  // Constants
  private static final long serialVersionUID = 99711266972132835 6L;

  // Member variables
  private ArrayList<String> buffer = new ArrayList<String>();
  private JTextArea logArea = new JTextArea();
  private int lineLimit = 0;
  private boolean hold = false;
  private boolean autoScroll = true;
  private boolean showTimestamp = true;
```

```
/**
 * Constructs a new LogArea.
 */
public LogArea()
{
  setLayout(new BorderLayout());
  logArea.setEditable(false);
  JScrollPane jsp = new JScrollPane(logArea);
  add(jsp, BorderLayout.CENTER);
}



/**
 * Constructs a new LogArea with the specified line limit.
 */
public LogArea(int _lineLimit)
{
  this();
  lineLimit = _lineLimit;
}



/**
 * Sets automatic scrolling.
 * @param _autoScroll If true, the text area will scroll
    automatically upon
 *            appending text to it.
 */
public void setAutoScroll(boolean _autoScroll)
{
  autoScroll = _autoScroll;
}



/**
 * Sets the hold state.
 * @param _hold If true, the text area will not be updated.
 */
public synchronized void setHold(boolean _hold)
{
  hold = _hold;

  if (!hold)
  {
    StringBuffer bufferedText = new StringBuffer();
    while (!buffer.isEmpty())
      bufferedText.append(buffer.remove(0));

    doAppend(bufferedText.toString());
  }
}



/**
 * Sets the line limit of this LogArea.
```

```
   * @param _lineLimit The new line limit.
   */
  public synchronized void setLineLimit(int _lineLimit)
  {
    lineLimit = _lineLimit;
  }



  /**
   * Enables or disables the timestamp.
   * @param _showTimestamp If true, a timestamp will be appended to
      the
   *            beginning of each message.
   */
  public void setShowTimestamp(boolean _showTimestamp)
  {
    showTimestamp = _showTimestamp;
  }



  /**
   * Appends the specified text into the LogArea, buffering if hold
      is on.
   *
   * @param text The text to append to the log.
   */
  public synchronized void append(String text)
  {
    // Construct line
    StringBuffer line = new StringBuffer();
    if (showTimestamp)
      line.append("<" + new Date().toString() + "> ");
    line.append(text + "\n");

    if (hold)
    {
      // Crop buffer
      if (lineLimit > 0 && buffer.size() > lineLimit)
        buffer.remove(0);

      // Buffer line
      buffer.add(line.toString());
    }
    else
      doAppend(line.toString());
  }



  /**
   * Appends the specified text into the LogArea.
   *
   * @param text The text to append to the log.
   */
  private void doAppend(String text)
  {
    // Append line to text area
    logArea.append(text);

    // Crop text area text
```

```
    if (lineLimit > 0)
    {
      int currentLines = logArea.getLineCount() - 1;
      if (currentLines > lineLimit)
      {
        try
        {
          int offset = logArea.getLineStartOffset(currentLines -
              lineLimit);
          logArea.setText(logArea.getText().substring(offset));
        }
        catch (BadLocationException e)
        {
          e.printStackTrace();
        }
      }
    }

    // Scroll down
    if (autoScroll)
      logArea.setCaretPosition(logArea.getDocument().getLength());
  }



  /**
   * Returns the number of chatacters in the LogArea.
   *
   * @return The number of chatacters in the LogArea.
   */
  public int getTextSize()
  {
    return logArea.getDocument().getLength();
  }



  /**
   * Copies the contents of the LogArea into the system clipboard.
   */
  public void copy()
  {
    Toolkit.getDefaultToolkit().getSystemClipboard().setContents(new
        StringSelection(logArea.getText()), null);
  }



  /**
   * Clears the contents of the LogArea.
   */
  public void clear()
  {
    buffer.clear();
    logArea.setText("");
  }
}
```

C.7 Class *TableModelStatistics*

```
import java.text.DecimalFormat;

import javax.swing.table.AbstractTableModel;

import jeva.ga.Parameters;



public class TableModelStatistics extends AbstractTableModel
{
  private static final String[] ROW_NAMES = new String[] { "
     Generation", "Stable Generations", "Mean Fitness", "Best
     Fitness", "Overall Best Fitness", "Mutation Rate", "Crossover
     Rate", "Consulted By", "Sexed By", "Sexless" };
  private static final DecimalFormat formatter = new DecimalFormat("
     0.000E0");
  private AgentSolver solver = null;



  public TableModelStatistics(AgentSolver _solver)
  {
    solver = _solver;
  }



  public boolean isCellEditable(int rowIndex, int columnIndex)
  {
    return false;
  }



  public int getColumnCount()
  {
    return 2;
  }



  public int getRowCount()
  {
    return ROW_NAMES.length;
  }



  public Object getValueAt(int rowIndex, int columnIndex)
  {
    if (columnIndex == 0)
      return ROW_NAMES[rowIndex];

    else if ((columnIndex == 1) && (solver.getBreeder() != null))
    {
      if (rowIndex == 0)
        return solver.getBreeder().getGeneration();
      else if (rowIndex == 1)
        return solver.getBreeder().getStableGenerations();
```

```
      else if (rowIndex == 2)
        return solver.getBreeder().getLastPopulation() == null ? "N/
            A" : formatter.format(solver.getBreeder().
            getLastPopulation().getFitnessMean());
      else if (rowIndex == 3)
        return solver.getBreeder().getBestPopulation() == null ? "N/
            A" : formatter.format(solver.getBreeder().
            getLastPopulation().getFitnessBest());
      else if (rowIndex == 4)
        return solver.getBreeder().getBestPopulation() == null ? "N/
            A" : formatter.format(solver.getBreeder().
            getBestPopulation().getFitnessBest());
      else if (rowIndex == 5)
        return formatter.format(solver.getBreeder().getParameters().
            getDouble(Parameters.MUTATION_RATE));
      else if (rowIndex == 6)
        return formatter.format(solver.getBreeder().getParameters().
            getDouble(Parameters.CROSSOVER_RATE));
      else if (rowIndex == 7)
        return solver.getTotalConsultedBy();
      else if (rowIndex == 8)
        return solver.getTotalSexedBy();
      else if (rowIndex == 9)
        return solver.getBreeder().getGeneration() - solver.
            getLastSex();
    }

    return null;
  }
}
```

C.8 Class *TableModelHistory*

```
import java.io.Serializable;
import java.text.DecimalFormat;

import javax.swing.table.AbstractTableModel;



public class TableModelHistory extends AbstractTableModel
{
  private static final DecimalFormat formatter = new DecimalFormat("
      0.0E0");
  private AgentSolver solver = null;



  public TableModelHistory(AgentSolver _solver)
  {
    solver = _solver;
  }



  public boolean isCellEditable(int rowIndex, int columnIndex)
  {
    return false;
```

```
  }


  public int getColumnCount()
  {
    return 5;
  }


  public int getRowCount()
  {
    return solver.listAllKnownPeers().size();
  }


  public Object getValueAt(int rowIndex, int columnIndex)
  {
    int currentColumnIndex = 0;

    if (columnIndex == currentColumnIndex++)
      return solver.listAllKnownPeers().get(rowIndex);

    else if ((columnIndex == currentColumnIndex++) && (solver.
        getBreeder() != null))
    {
      Serializable key = (Serializable)getValueAt(rowIndex, 0);
      return formatter.format(solver.getHistoryFor(key));
    }

    else if ((columnIndex == currentColumnIndex++) && (solver.
        getBreeder() != null))
    {
      Serializable key = (Serializable)getValueAt(rowIndex, 0);
      return formatter.format(solver.getSuggestionsFor(key));
    }

    else if ((columnIndex == currentColumnIndex++) && (solver.
        getBreeder() != null))
    {
      Serializable key = (Serializable)getValueAt(rowIndex, 0);
      return formatter.format(solver.getMemoryFor(key));
    }

    else if ((columnIndex == currentColumnIndex++) && (solver.
        getBreeder() != null))
    {
      Serializable key = (Serializable)getValueAt(rowIndex, 0);
      return formatter.format(solver.getRankingFor(key));
    }

    return null;
  }
}
```

C.9 Class *TableModelCounts*

```
import java.io.Serializable;

import javax.swing.table.AbstractTableModel;



public class TableModelCounts extends AbstractTableModel
{
  //private static final DecimalFormat formatter = new DecimalFormat
      ("0.0E0");
  private AgentSolver solver = null;



  public TableModelCounts(AgentSolver _solver)
  {
    solver = _solver;
  }



  public boolean isCellEditable(int rowIndex, int columnIndex)
  {
    return false;
  }



  public int getColumnCount()
  {
    return 6;
  }



  public int getRowCount()
  {
    return solver.listAllKnownPeers().size();
  }



  public Object getValueAt(int rowIndex, int columnIndex)
  {
    int currentColumnIndex = 0;

    if (columnIndex == currentColumnIndex++)
      return solver.listAllKnownPeers().get(rowIndex);

    else if ((columnIndex == currentColumnIndex++) && (solver.
        getBreeder() != null))
    {
      Serializable key = (Serializable)getValueAt(rowIndex, 0);
      return solver.getCountConsultedFor(key);
    }

    else if ((columnIndex == currentColumnIndex++) && (solver.
        getBreeder() != null))
    {
      Serializable key = (Serializable)getValueAt(rowIndex, 0);
      return solver.getCountConsultedByFor(key);
```

```
    }

    else if ((columnIndex == currentColumnIndex++) && (solver.
       getBreeder() != null))
    {
      Serializable key = (Serializable)getValueAt(rowIndex, 0);
      return solver.getCountSexedFor(key);
    }

    else if ((columnIndex == currentColumnIndex++) && (solver.
       getBreeder() != null))
    {
      Serializable key = (Serializable)getValueAt(rowIndex, 0);
      return solver.getCountSexedByFor(key);
    }

    else if ((columnIndex == currentColumnIndex++) && (solver.
       getBreeder() != null))
    {
      Serializable key = (Serializable)getValueAt(rowIndex, 0);
      return solver.getCountSuggestedFor(key);
    }

    return null;
  }
}
```

C.10 Class *TableCellRendererHistory*

```
import java.awt.Color;
import java.awt.Component;
import java.io.Serializable;

import javax.swing.JLabel;
import javax.swing.JTable;
import javax.swing.table.DefaultTableCellRenderer;



public class TableCellRendererHistory extends
    DefaultTableCellRenderer
{
  private AgentSolver solver;



  public TableCellRendererHistory(AgentSolver _solver)
  {
    solver = _solver;
  }



  @SuppressWarnings("static-access")
  public Component getTableCellRendererComponent(JTable table,
      Object value, boolean isSelected, boolean hasFocus, int row,
      int column)
  {
```

```
    Serializable id = (Serializable)table.getValueAt(row, 0);

    JLabel renderer = (JLabel)super.getTableCellRendererComponent(
       table, value, isSelected, hasFocus, row, column);

    if (solver.listOfDamagedAgentsCheat.contains(id))
      renderer.setForeground(Color.RED);
    else
      renderer.setForeground(Color.BLACK);

    return renderer;
  }
}
```

C.11 Class *GraphFitness*

```
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Stroke;

import javax.swing.JPanel;


import jeva.ga.Breeder;



public class GraphFitness extends JPanel
{
  public static final Stroke STROKE_SOLID = new BasicStroke(1.0f);
  public static final Stroke STROKE_DASH = new BasicStroke(1.0f,
      BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER, 10.0f, new float
      [] { 4.0f }, 0.0f);

  public static final int GRID_SPACING_X = 20;
  public static final int GRID_SPACING_Y = 20;

  private Breeder breeder = null;
  private Double tfit = null;
  private Double cycle = null;
  private boolean damaged;



  public GraphFitness(Breeder _breeder, Double _tfit, Double _cycle,
      boolean _damaged)
  {
    breeder = _breeder;
    tfit = _tfit;
    cycle = _cycle;
    damaged = _damaged;
  }



  public void paintComponent(Graphics g)
  {
```

```
super.paintComponent(g);

Graphics2D g2 = (Graphics2D)g;

// Check if there are enough points to plot
int historyGens = breeder.getFitnessHistoryBest().size();
if ((breeder == null) || (historyGens < 1))
  return;

// Determine limits
//double fitnessMin = Utilities.findMin((Double[])breeder.
    getFitnessHistoryBest().toArray(new Double[0]));
double fitnessMin = breeder.getBestPopulation().getFitnessBest()
    ;
double fitnessMax = Utilities.findMax((Double[])breeder.
    getFitnessHistoryWorst().toArray(new Double[0]));
//    double fitnessMin = 10;
//    double fitnessMax = 1200;
double scaleX = (double)getWidth() / (double)historyGens;
int offsetX = 0;
double scaleY = -1 * getHeight() / (fitnessMax - fitnessMin);
int offsetY = getHeight() - (int)(fitnessMin * scaleY);

// Draw cycle limits
g2.setColor(Color.GREEN);
g2.setStroke(STROKE_SOLID);
int genOfFirstHistoryEntry = breeder.getGeneration() -
    historyGens;
for (int i = 0; i < historyGens; i++)
  if ((genOfFirstHistoryEntry + i) % cycle == 0)
  {
    int x = offsetX + (int)(scaleX * i);
    g2.drawLine(x, 0, x, getHeight());
  }

// Draw best & worst fitness graph
g2.setColor(damaged ? Color.RED : Color.GRAY);
g2.setStroke(STROKE_SOLID);
int lastX = offsetX;
int lastYW = offsetY + (int)(scaleY * breeder.
    getFitnessHistoryWorst().get(0));
int lastYB = offsetY + (int)(scaleY * breeder.
    getFitnessHistoryBest().get(0));
for (int i = 1; i < historyGens; i++)
{
  int x = offsetX + (int)(scaleX * i);
  int yW = offsetY + (int)(scaleY * breeder.
      getFitnessHistoryWorst().get(i));
  int yB = offsetY + (int)(scaleY * breeder.
      getFitnessHistoryBest().get(i));
  g2.fillPolygon(new int[] { lastX, x, x, lastX }, new int[] {
      lastYW, yW, yB, lastYB }, 4);
  lastX = x;
  lastYW = yW;
  lastYB = yB;
}

// Draw mean fitness graph
g2.setColor(Color.LIGHT_GRAY);
g2.setStroke(STROKE_SOLID);
lastX = offsetX;
int lastY = offsetY + (int)(scaleY * breeder.
```

```
        getFitnessHistoryMean().get(0));
    for (int i = 1; i < historyGens; i++)
    {
      int x = offsetX + (int)(scaleX * i);
      int y = offsetY + (int)(scaleY * breeder.getFitnessHistoryMean
          ().get(i));
      g2.drawLine(lastX, lastY, x, y);
      lastX = x;
      lastY = y;
    }

    // Draw target fitness line
    if (tfit != null)
    {
      g2.setColor(Color.RED);
      g2.setStroke(STROKE_DASH);
      int targetY = offsetY + (int)(scaleY * tfit);
      g2.drawLine(0, targetY, getWidth(), targetY);
    }

    // Draw best fitness line
    g2.setColor(Color.RED);
    g2.setStroke(STROKE_DASH);
    int bestY = offsetY + (int)(scaleY * breeder.getBestPopulation()
        .getFitnessBest());
    bestY -= 1;
    g2.drawLine(0, bestY, getWidth(), bestY);

    // Draw stable generations line
    g2.setColor(Color.RED);
    g2.setStroke(STROKE_DASH);
    int stableX = offsetX + (int)(scaleX * (historyGens - breeder.
        getStableGenerations()));
    stableX = Math.min(stableX, getWidth() - 1);
    g2.drawLine(stableX, 0, stableX, getHeight());
  }
}
```

C.12 Class *Options*

```
import java.util.HashMap;



public class Options
{
  private HashMap<String, String> options = new HashMap<String,
      String>();



  public Options(String[] args)
  {
    options = Utilities.parseOptions(args);
  }



  public void put(String key, String value)
```

```
  {
    options.put(key, value);
  }



  public Boolean getBoolean(String key)
  {
    return Boolean.parseBoolean(options.get(key));
  }



  public Integer getInteger(String key)
  {
    return (options.containsKey(key) ? Integer.parseInt(options.get(
        key)) : null);
  }



  public Double getDouble(String key)
  {
    return (options.containsKey(key) ? Double.parseDouble(options.
        get(key)) : null);
  }



  public String getString(String key)
  {
    return options.get(key);
  }



  public String toString()
  {
    return options.toString();
  }
}
```

C.13 Class *Utilities*

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Random;
import java.util.StringTokenizer;



public class Utilities
{
  public static final Random RNG = new Random();



  public static Object createObject(String className)
  {
```

```
    try
    {
      Class<?> theClass = (Class<?>)Class.forName(className);
      return theClass.newInstance();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }

    return null;
  }



  public static Integer[] chooseRandom(int min, int max, int n)
  {
    ArrayList<Integer> idAll = new ArrayList<Integer>();
    ArrayList<Integer> idSubset = new ArrayList<Integer>();
    for (int i = min; i < max; i++)
      idAll.add(i);
    for (int i = 0; i < n; i++)
      idSubset.add(idAll.remove(RNG.nextInt(idAll.size())));

    return idSubset.toArray(new Integer[0]);
  }



  public static Double[] parseArrayDouble(String s)
  {
    if (!s.startsWith("{") || !s.endsWith("}"))
      return new Double[0];

    s = s.substring(1, s.length() - 1);
    StringTokenizer st = new StringTokenizer(s, ";");
    if (!st.hasMoreElements())
      return new Double[0];

    ArrayList<Double> list = new ArrayList<Double>();
    while (st.hasMoreElements())
    {
      try
      {
        list.add(Double.parseDouble(st.nextToken()));
      }
      catch (NumberFormatException e)
      {
        return new Double[0];
      }
    }

    return list.toArray(new Double[0]);
  }



  public static boolean isArrayDouble(String s)
  {
    if (!s.startsWith("{") || !s.endsWith("}"))
      return false;
```

```
    s = s.substring(1, s.length() - 1);
    StringTokenizer st = new StringTokenizer(s, ";");
    if (!st.hasMoreElements())
      return false;

    while (st.hasMoreElements())
    {
      try
      {
        Double.parseDouble(st.nextToken());
      }
      catch (NumberFormatException e)
      {
        return false;
      }
    }

    return true;
  }



  public static boolean isDouble(String s)
  {
    try
    {
      Double.parseDouble(s);
    }
    catch (NumberFormatException e)
    {
      return false;
    }

    return true;
  }



  /**
   * Sleep for the specified number of milliseconds
   * @param millis The number of milliseconds to sleep.
   */
  public static void sleep(int millis)
  {
    try
    {
      Thread.sleep(millis);
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }



  /**
   * Searches in an array for the maximum value.
   * @param array The array.
   * @return The maximum value.
   */
  public static double findMax(Double[] array)
```

```
{
  double max = Double.NEGATIVE_INFINITY;

  for (int i = 0; i < array.length; i++)
    if (array[i] > max)
      max = array[i];

  return max;
}



/**
 * Searches in an array for the minimum value.
 * @param array The array.
 * @return The minimum value.
 */
public static double findMin(Double[] array)
{
  double min = Double.POSITIVE_INFINITY;

  for (int i = 0; i < array.length; i++)
    if (array[i] < min)
      min = array[i];

  return min;
}



/**
 * Limits a value between the specified lower and upper limits.
 * @param value The value to limit.
 * @param minimum The minimum value to limit.
 * @param maximum The maximum value to limit.
 * @return The value, if it is between the limits; minimum, if it
    is smaller; or maximum, if it is bigger
 */
public static int limit(int value, int minimum, int maximum)
{
  value = Math.min(value, maximum);
  value = Math.max(value, minimum);
  return value;
}



/**
 * Limits a value between the specified lower and upper limits.
 * @param value The value to limit.
 * @param minimum The minimum value to limit.
 * @param maximum The maximum value to limit.
 * @return The value, if it is between the limits; minimum, if it
    is smaller; or maximum, if it is bigger
 */
public static double limit(double value, double minimum, double
   maximum)
{
  value = Math.min(value, maximum);
  value = Math.max(value, minimum);
  return value;
}
```

```
  /**
   * Parses the specified command-line arguments and returns a
       HashMap with key-value pairs. Each key (parameter name) must
       start with a dash (-), while parameter values must not.
   * @param args The command-line arguments.
   * @return The HashMap with the key-value pairs.
   */
  public static HashMap<String, String> parseOptions(String[] args)
  {
    HashMap<String, String> options = new HashMap<String, String>();

    for (int i = 0; i < args.length; i++)
      if (args[i].startsWith("-"))
      {
        String key = null;
        String value = null;

        if (args[i].contains("="))
        {
          key = args[i].substring(1, args[i].indexOf("="));
          value = args[i].substring(args[i].indexOf("=") + 1, args[i
              ].length());
        }
        else
          key = args[i].substring(1);

        options.put(key, value);
      }

    return options;
  }
}
```

C.14 Class *Constants*

```
public interface Constants
{
  // User option keys
  public static final String OPTION_PROTOCOL = "protocol";
  public static final String OPTION_N = "n";
  public static final String OPTION_ASK = "ask";
  public static final String OPTION_DMG = "dmg";
  public static final String OPTION_CYCLE = "cycle";
  public static final String OPTION_TYPE = "type";
  public static final String OPTION_CUMULATIVE = "cumulative";
  //  public static final String OPTION_AVERAGE = "average";
  public static final String OPTION_NVARS = "nvars";
  public static final String OPTION_NBITS = "nbits";
  public static final String OPTION_TGEN = "tgen";
  public static final String OPTION_TFIT = "tfit";
  public static final String OPTION_PS = "ps";
  public static final String OPTION_EP = "ep";
  public static final String OPTION_CR = "cr";
  public static final String OPTION_MR = "mr";
  public static final String OPTION_IF = "if";
  public static final String OPTION_SF = "sf";
  public static final String OPTION_CF = "cf";
```

```
  public static final String OPTION_MF = "mf";
  public static final String OPTION_EF = "ef";
  public static final String OPTION_OBJ = "obj";
  public static final String OPTION_GUI = "gui";
  public static final String OPTION_MONITOR = "monitor";
  public static final String OPTION_CHATTER = "chatter";
  public static final String OPTION_PROGRESS = "progress";

  // Fixed options
  public static final int N_EVALUATION_THREADS = 1;
  public static final int HISTORY_LENGTH = 100; //30  //10000
  public static final double MIN_MR = 1e-5; //1e-5
}
```

Bibliography

[Abbass, 2002] Abbass, H. A. (2002). An evolutionary artificial neural networks approach for breast cancer diagnosis. *Artificial Intelligence in Medicine*, 25(3):265–281.

[Abdul-Rahman and Hailes, 2000] Abdul-Rahman, A. and Hailes, S. (2000). Supporting trust in virtual communities. In *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 6 - Volume 6*, HICSS '00, pages 6007–, Washington, DC, USA. IEEE Computer Society.

[Alba and Troya, 1999] Alba, E. and Troya, J. M. (1999). A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52.

[Allan, 2009] Allan, R. (2009). Survey of agent based modelling and simulation tools. Technical report, Computational Science and Engineering Department, STFC Daresbury Laboratory.

[Ammar and Tao, 2000] Ammar, H. H. and Tao, Y. (2000). Fingerprint registration using genetic algorithms. In *Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'00)*, ASSET '00, pages 148–, Washington, DC, USA. IEEE Computer Society.

[Anderson, 2004] Anderson, D. P. (2004). Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA. IEEE Computer Society.

[Androutsellis-Theotokis and Spinellis, 2004] Androutsellis-Theotokis, S. and Spinellis, D. (2004). A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36:335–371.

[Arenas et al., 2002] Arenas, M. G., Collet, P., Eiben, A. E., Jelasity, M., Guervós, J. J. M., Paechter, B., Preuß, M., and Schoenauer, M. (2002). A framework for distributed evolutionary algorithms. In *PPSN VII: Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, pages 665–675, London, UK. Springer-Verlag.

[Ashraf et al., 2012] Ashraf, R. A., Luna, F., Dechev, D., and DeMara, R. F. (2012). Designing digital circuits for fpgas using parallel genetic algorithms (wip). In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, TMS/DEVS '12, pages 15:1–15:6, San Diego, CA, USA. Society for Computer Simulation International.

[Assudani and Malik, 2012] Assudani, P. J. and Malik, L. G. (2012). Article: Genetic algorithm based dot pattern image processing. *IJCA Proceedings on National Conference on Innovative Paradigms in Engineering and Technology (NCIPET 2012)*, ncipet(14):31–35. Published by Foundation of Computer Science, New York, USA.

[Back, 1992] Back, T. (1992). Self-adaptation in genetic algorithms. In *Proceedings of the First European Conference on Artificial Life*, pages 263–271. MIT Press.

[Barber and Kim, 2003] Barber, K. S. and Kim, J. (2003). Soft security: isolating unreliable agents from society. In *Proceedings of the 2002 international conference on Trust, reputation, and security: theories and practice*, AAMAS'02, pages 224–233, Berlin, Heidelberg. Springer-Verlag.

[Barricelli, 1962] Barricelli, N. A. (1962). Numerical testing of evolution theories. *Acta Biotheoretica*, 16(1-2).

[Belding, 1995] Belding, T. C. (1995). The distributed genetic algorithm revisited. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 114–121, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Berndt and Watkins, 2005] Berndt, D. J. and Watkins, A. (2005). High volume software testing using genetic algorithms. In *Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences - Volume 09*, HICSS '05, pages 318.2–, Washington, DC, USA. IEEE Computer Society.

[Bondi, 2000] Bondi, A. B. (2000). Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, WOSP '00, pages 195–203, New York, NY, USA. ACM.

[Brzykcy, 2009] Brzykcy, G. (2009). Information flow in a peer-to-peer data integration system. In *Proceedings of the Third KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, KES-AMSTA '09, pages 420–429, Berlin, Heidelberg. Springer-Verlag.

[Burnett et al., 2011] Burnett, C., Norman, T. J., and Sycara, K. (2011). Trust decision-making in multi-agent systems. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume One*, IJCAI'11, pages 115–120. AAAI Press.

[Cantu-Paz, 1998] Cantu-Paz, E. (1998). A survey of parallel genetic algorithms. *Calculateurs Paralleles*, 102.

[Caron et al., 2009] Caron, E., Desprez, F., Petit, F., and Tedeschi, C. (2009). Peer-to-peer service discovery for grid computing. In Antonopoulos, N., Exarchakos, G., Li, M., and Liotta, A., editors, *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications*. IGI Global, Information Science Publishing. INT LIP6 Regal.

[Chalamalasetti et al., 2009] Chalamalasetti, S., Purohit, S., Margala, M., and Vanderbauwhede, W. (2009). Mora - an architecture and programming model for a resource efficient coarse grained reconfigurable processor. In *2009 NASA/ESA Conference on Adaptive Hardware and Systems, 29 July 2009 - 1 Aug. 2009, San Francisco, CA, USA*, pages 389–396. IEEE Computer Society, Piscataway, N.J., USA.

[Chatzinikolaou, 2003] Chatzinikolaou, N. (2003). Evolving neural controllers for a simulated lander. Master's thesis, University of Sussex, Brighton, UK.

[Chatzinikolaou, 2010] Chatzinikolaou, N. (2010). Coordinating evolution - designing a self-adapting distributed genetic algorithm. In Filipe, J. and Cordeiro, J., editors, *ICEIS (2)*, pages 13–20. SciTePress.

[Chatzinikolaou and Robertson, 2012] Chatzinikolaou, N. and Robertson, D. (2012). The use of reputation as noise-resistant selection bias in a co-evolutionary multi-agent system. In Soule, T. and Moore, J. H., editors, *GECCO*, pages 983–990. ACM.

[Chow et al., 2009] Chow, A. L., Golubchik, L., and Misra, V. (2009). Bittorrent: An extensible heterogeneous model. In *Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Infocom)*, Rio de Janeiro.

[Clune et al., 2005] Clune, J., Goings, S., Punch, B., and Goodman, E. (2005). Investigations in meta-GAs: panaceas or pipe dreams? In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 235–241, New York, NY, USA. ACM.

[Conte and Paolucci, 2002] Conte, R. and Paolucci, M. (2002). *Reputation in Artificial Societies: Social Beliefs for Social Order (Multiagent Systems, Artificial Societies, and Simulated Organizations)*. Springer.

[Cruz and Ducasse, 1999] Cruz, J. C. and Ducasse, S. (1999). Coordinating open distributed systems. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, FTDCS '99, pages 125–, Washington, DC, USA. IEEE Computer Society.

[Darwin, 1870] Darwin, C. (1870). The descent of man, and selection in relation to sex. Freeman #936.

[Darwin, 1872] Darwin, C. (1872). *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. J. Murray, sixth edition.

[Das and Vemuri, 2007] Das, A. and Vemuri, R. (2007). An automated passive analog circuit synthesis framework using genetic algorithms. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, ISVLSI '07, pages 145–152, Washington, DC, USA. IEEE Computer Society.

[De Jong, 1975] De Jong, K. (1975). *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan.

[Deguet et al., 2007] Deguet, J., Magnin, L., and Demazeau, Y. (2007). Emergence and software development based on a survey of emergence definitions. In Namatame, A., Kurihara, S., and Nakashima, H., editors, *Emergent Intelligence of Networked Agents*, volume 56 of *Studies in Computational Intelligence*, pages 13–21. Springer Berlin / Heidelberg.

[Dorigo and Stützle, 2004] Dorigo, M. and Stützle, T. (2004). *Ant colony optimization*. Bradford Books. MIT Press.

[Douceur, 2002] Douceur, J. R. (2002). The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK. Springer-Verlag.

[Dreżewski et al., 2009] Dreżewski, R., Woźniak, P., and Siwik, L. (2009). Agent-based evolutionary system for traveling salesman problem. In *Proceedings of the 4th International Conference on Hybrid Artificial Intelligence Systems*, HAIS '09, pages 34–41, Berlin, Heidelberg. Springer-Verlag.

[Du and Fu, 2011] Du, F. and Fu, F. (2011). Partner selection shapes the strategic and topological evolution of cooperation. *Dynamic Games and Applications*, 1(3):354–369.

[Dunbar, 1998] Dunbar, R. I. M. (1998). The social brain hypothesis. *Evolutionary Anthropology: Issues, News, and Reviews*, 6(5):178–190.

[Eiben et al., 2000] Eiben, A. E., Hinterding, R., Hinterding, A. E. E. R., and Michalewicz, Z. (2000). Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3:124–141.

[Eiben et al., 2006] Eiben, A. E., Schut, M. C., and Wilde, A. R. D. (2006). Boosting genetic algorithms with self-adaptive selection. In *In Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1584–1589.

[Espejo et al., 2010] Espejo, P. G., Ventura, S., and Herrera, F. (2010). A survey on the application of genetic programming to classification. *Trans. Sys. Man Cyber Part C*, 40(2):121–144.

[Esteva et al., 2000] Esteva, M., Rodrguez-Aguilar, J. A., Arcos, J. L., Sierra, C., and Garcia, P. (2000). Institutionalising open multi-agent systems. Technical report, Artificial Intelligence Research Institute. Spanish Council for Scientific Research. IIIA Research Report 2000-01 (http://- www.iiia.csic.es/Publications/Reports/2000.

[Falcone and Castelfranchi, 2001] Falcone, R. and Castelfranchi, C. (2001). *Trust and Deception in Virtual Societies*, chapter Social Trust: A Cognitive Approach, pages 55–90. Kluwer Academic Publishers.

[Fogel, 1962] Fogel, L. J. (1962). Toward inductive inference automata. In *IFIP Congress'62*, pages 395–400.

[Fortino and Russo, 2008] Fortino, G. and Russo, W. (2008). Using p2p, grid and agent technologies for the development of content distribution networks. *Future Gener. Comput. Syst.*, 24:180–190.

[Foster and Iamnitchi, 2003] Foster, I. and Iamnitchi, A. (2003). On death, taxes, and the convergence of peer-to-peer and grid computing. In *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS03*, pages 118–128.

[Foster et al., 2001] Foster, I., Kesselman, C., and Tuecke, S. (2001). The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15:200–222.

[Foster et al., 2008] Foster, I., Zhao, Y., Raicu, I., and Lu, S. (2008). Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10.

[Georgiou et al., 2005] Georgiou, C., Kowalski, D. R., and Shvartsman, A. A. (2005). Efficient gossip and robust distributed computation. *Theor. Comput. Sci.*, 347:130–166.

[Ghosh and Mitchell, 2006] Ghosh, P. and Mitchell, M. (2006). Segmentation of medical images using a genetic algorithm. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, pages 1171–1178, New York, NY, USA. ACM.

[Glickman and Sycara, 2000] Glickman, M. R. and Sycara, K. (2000). Reasons for premature convergence of self-adapting mutation rates. In *In Proc. of the 2000 Congress on Evolutionary Computation*, pages 62–69.

[Goldberg, 1989] Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA.

[Gómez Mármol et al., 2011] Gómez Mármol, F., Martínez Pérez, G., and Gómez Marín-Blázquez, J. (2011). META-TACS: a Trust Model Demonstration of Robustness through a Genetic Algorithm. *Intelligent Automation and Soft Computing (Autosoft) Journal*, 17(1):41–59.

[Gondro and Kinghorn, 2007] Gondro, C. and Kinghorn, B. P. (2007). A simple genetic algorithm for multiple sequence alignment. *Genetics and molecular research GMR*, 6(4):964–982.

[Gonalves et al., 2002] Gonalves, J. F., Dr, R., Frias, R., Jos, J., Mendes, M., and Resende, M. G. C. (2002). A hybrid genetic algorithm for the job shop scheduling problem. *European Journal of Operational Research*, 167:2005.

[Gordon et al., 1999] Gordon, V. S., Pirie, R., Wachter, A., and Sharp, S. (1999). Terrain-based genetic algorithm (TBGA): Modeling parameter space as terrain. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 229–235, Orlando, Florida, USA. Morgan Kaufmann.

[Han, 2004] Han, P. (2004). A scalable P2P recommender system based on distributed collaborative filtering. *Expert Systems with Applications*, 27(2):203–210.

[Hazard and Singh, 2010] Hazard, C. J. and Singh, M. P. (2010). An architectural approach to combining trust and reputation. In *Proceedings of the 13th AAMAS Workshop on Trust in Agent Societies (Trust)*.

[Herzog et al., 2009] Herzog, A., Handrich, S., and Herrmann, C. (2009). Multi-objective parameter estimation of biologically plausible neural networks in different behavior stages. In *2009 IEEE Congress on Evolutionary Computation (CEC'2009)*, pages 793–799, Trondheim, Norway. IEEE Press.

[Hesser and Männer, 1991] Hesser, J. and Männer, R. (1991). Towards an optimal mutation probability for genetic algorithms. In *PPSN I: Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, pages 23–32, London, UK. Springer-Verlag.

[Hoffmeister and Bck, 1991] Hoffmeister, F. and Bck, T. (1991). Genetic algorithms and evolution strategies: Similarities and differences. In Schwefel, H.-P. and Mnner, R., editors, *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*, pages 455–469. Springer Berlin / Heidelberg. 10.1007/BFb0029787.

[Holland, 1975] Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.

[Hong et al., 2005] Hong, C., Kim, W., and Kim, Y. (2005). Distributed channel routing using genetic algorithm. In Liew, K.-M., Shen, H., See, S., Cai, W., Fan, P., and Horiguchi, S., editors, *Parallel and Distributed Computing: Applications and*

Technologies, volume 3320 of *Lecture Notes in Computer Science*, pages 73–83. Springer Berlin / Heidelberg.

[Horling and Lesser, 2004] Horling, B. and Lesser, V. (2004). A survey of multi-agent organizational paradigms. *Knowl. Eng. Rev.*, 19:281–316.

[Huang, 2010] Huang, L. (2010). Large scale cooperative multiagent system based on semantic p2p network. In *Proceedings of the 2010 First International Conference on Networking and Distributed Computing*, ICNDC '10, pages 381–386, Washington, DC, USA. IEEE Computer Society.

[Hübner et al., 2008] Hübner, J. F., Vercouter, L., and Boissier, O. (2008). Instrumenting Multi-Agent Organisations with Reputation Artifacts. In Dignum, V. and Matson, E., editors, *Coordination, Organizations, Institutions and Norms (COIN@AAAI), held with AAAI 2008*, pages 17–24, Chicago, United States. AAAI Press.

[Huynh et al., 2006] Huynh, T., Jennings, N. R., and Shadbolt, N. (2006). An integrated trust and reputation model for open multi-agent systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 13(2):119–154.

[Ismail and Josang, 2002] Ismail, R. and Josang, A. (2002). The beta reputation system. In *Proceedings of the 15th Bled Conference on Electronic Commerce*.

[Jaffar et al., 2007] Jaffar, J., Yap, R. H. C., and Zhu, K. Q. (2007). Generalized committed choice. In Murphy, A. L. and Vitek, J., editors, *Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4467 of *Lecture Notes in Computer Science*, pages 191–210. Springer.

[Jennings, 2001] Jennings, N. R. (2001). An agent-based approach for building complex software systems. *Commun. ACM*, 44:35–41.

[Krink and Ursem, 2000] Krink, T. and Ursem, R. K. (2000). Parameter control using the agent based patchwork model. In *Proceedings of the Congress on Evolutionary Computation*, pages 77–83.

[Kureichik et al., 2009] Kureichik, V. M., Malioukov, S. P., Kureichik, V. V., and Malioukov, A. S. (2009). *Genetic Algorithms for Applied CAD Problems*. Springer Publishing Company, Incorporated, 1st edition.

[Larson et al., 2009] Larson, S. M., Snow, C. D., Shirts, M., and Pande, V. S. (2009). Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*.

[Law and Szeto, 2007] Law, N. L. and Szeto, K. Y. (2007). Adaptive genetic algorithm with mutation and crossover matrices. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pages 2330–2333, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Li et al., 2004] Li, Y., Ang, K., Chong, G., Feng, W., Tan, K., and Kashiwagi, H. (2004). Cautocsd-evolutionary search and optimisation enabled computer automated control system design. *International Journal of Automation and Computing*, 1(1):76–88.

[Lim et al., 2007] Lim, D., Ong, Y.-S., Jin, Y., Sendhoff, B., and Lee, B.-S. (2007). Efficient hierarchical parallel genetic algorithms using grid computing. *Future Gener. Comput. Syst.*, 23(4):658–670.

[Luck et al., 2004] Luck, M., McBurney, P., and Preist, C. (2004). A manifesto for agent technology: Towards next generation computing. *Journal of Autonomous Agents and Multi-Agent Systems*, 9(3):203–252.

[Maniezzo and Carbonaro, 1999] Maniezzo, V. and Carbonaro, A. (1999). Ant colony optimization: An overview. In *Essays and Surveys in Metaheuristics*, pages 21–44. Kluwer Academic Publishers.

[Marmol et al., 2009] Marmol, F. G., Perez, G. M., and Skarmeta, A. F. G. (2009). TACS, a Trust Model for P2P Networks. *Wireless Personal Communications, Special Issue on "Information Security and data protection in Future Generation Communication and Networking"*.

[Marney et al., 2001] Marney, J., Fyfe, C., Tarbert, H., and Miller, D. (2001). Risk Adjusted Returns to Technical Trading Rules: a Genetic Programming Approach. Technical Report 147, Society for Computational Economics.

[Maron and Moore, 1997] Maron, O. and Moore, A. W. (1997). The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11:193–225.

[Meyer-Nieberg and Beyer, 2006] Meyer-Nieberg, S. and Beyer, H.-G. (2006). Self-adaptation in evolutionary algorithms. In *Parameter Setting in Evolutionary Algorithm*, pages 47–76. Springer.

[Miller et al., 1995] Miller, B. L., Miller, B. L., Goldberg, D. E., and Goldberg, D. E. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212.

[Miller, 2001] Miller, G. (2001). *The Mating Mind: How Sexual Choice Shaped the Evolution of Human Nature*. Anchor.

[Muchnick, 1997] Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[Munawar et al., 2008] Munawar, A., Wahib, M., Munetomo, M., and Akama, K. (2008). A survey: Genetic algorithms and the fast evolving world of parallel computing. *High Performance Computing and Communications, 10th IEEE International Conference*, pages 897–902.

[Murata et al., 2007] Murata, Y., Shibata, N., Yasumoto, K., Ito, M., and Words, K. (2007). Agent oriented self adaptive genetic algorithm.

[Nowostawski and Poli, 1999] Nowostawski, M. and Poli, R. (1999). Parallel genetic algorithm taxonomy. In *Proceedings of the Third International*, pages 88–92. IEEE.

[Okabe et al., 2005] Okabe, T., Jin, Y., and Sendhoff, B. (2005). *Theoretical Comparisons of Search Dynamics of Genetic Algorithms and Evolution Strategies*, volume 1, pages 382–389. IEEE Service Center.

[Owais et al., 2008] Owais, S. S. J., Snsel, V., Krmer, P., and Abraham, A. (2008). Survey: Using genetic algorithm approach in intrusion detection systems techniques. In Snsel, V., Abraham, A., Saeed, K., and Pokorn, J., editors, *CISIM*, pages 300–307. IEEE Computer Society.

[Page and Naughton, 2005] Page, A. J. and Naughton, T. J. (2005). Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing.

[Panti et al., 2002] Panti, M., Penserini, L., Spalazzi, L., and Tacconi, S. (2002). S.: A multi-agent system based on the p2p model to information integration. proposal to agentcities task force. In *Autonomous Agents and Multi-Agent Systems (AAMAS*.

[Paolucci and Conte, 2009] Paolucci, M. and Conte, R. (2009). *Reputation: Social Transmission for Partner Selection*, pages 243–260. Hershey: IGI Publishing.

[Paszkowicz, 2009] Paszkowicz, W. (2009). Genetic algorithms, a nature-inspired tool: Survey of applications in materials science and related fields. *Materials and Manufacturing Processes*, 24(2):174–197.

[P.N.Hrisheekesha and Sharma, 2010] P.N.Hrisheekesha and Sharma, J. (2010). Article: Evolutionary algorithm based optimal control in distribution system with dispersed generation. *International Journal of Computer Applications*, 1(14):31–37. Published By Foundation of Computer Science.

[Potter and Jong, 1994] Potter, M. A. and Jong, K. A. D. (1994). A cooperative coevolutionary approach to function optimization. pages 249–257. Springer-Verlag.

[Ramakrishna, 2002] Ramakrishna, R. S. (2002). A genetic algorithm for shortest path routing problem and the sizing of populations. *IEEE Transactions on Evolutionary Computation*, 6(6):566–579.

[Ramillien, 2001] Ramillien, G. (2001). Genetic algorithms for geophysical parameter inversion from altimeter data. *Geophysical Journal International*, 147(2):393–402.

[Rathore et al., 2011] Rathore, A., Bohara, A., Prashil, R. G., Prashanth, T. S. L., and Srivastava, P. R. (2011). Application of genetic algorithm and tabu search in software testing. In *Proceedings of the Fourth Annual ACM Bangalore Conference*, COMPUTE '11, pages 23:1–23:4, New York, NY, USA. ACM.

[Rechenberg, 1971] Rechenberg, I. (1971). *Evolutionsstrategie*. Frommann-Holzboog-Verlag.

[Ricordel and Demazeau, 2000] Ricordel, P.-M. and Demazeau, Y. (2000). From analysis to deployment: A multi-agent platform survey. In *Engineering Societies in the Agents World, LNAI 1972*, pages 93–105. Springer-Verlag.

[Ripeanu, 2001] Ripeanu, M. (2001). Peer-to-peer architecture case study: Gnutella network.

[Robertson, 2004a] Robertson, D. (2004a). International conference on logic programming. Sant-Malo, France.

[Robertson, 2004b] Robertson, D. (2004b). A lightweight coordination calculus for agent systems. In *In Declarative Agent Languages and Technologies*, pages 183–197.

[Robertson et al., 2006] Robertson, D., Giunchiglia, F., van Harmelen, F., Marchese, M., Sabou, M., Schorlemmer, M., Shadbolt, N., Siebes, R., Sierra, C., Walton, C., Dasmahapatra, S., Dupplaw, D., Lewis, P., Yatskevich, M., Kotoulas, S., de Pinninck, A. P., and Loizou, A. (2006). Open knowledge semantic webs through peer-to-peer interaction. Technical Report DIT-06-034, University of Trento.

[Robu and Holban, 2011] Robu, R. and Holban, S. (2011). A genetic algorithm for classification. In *Proceedings of the 2011 international conference on Computers and computing*, ICCC'11, pages 52–56, Stevens Point, Wisconsin, USA. World Scientific and Engineering Academy and Society (WSEAS).

[Rzevski and Skobelev, 2007] Rzevski, G. and Skobelev, P. (2007). Emergent intelligence in large scale multi-agent systems. *International Journal of Education and Information Technologies*, 1(2):64–71.

[Sabater and Sierra, 2001] Sabater, J. and Sierra, C. (2001). Regret: reputation in gregarious societies. In *Proceedings of the fifth international conference on Autonomous agents*, AGENTS '01, pages 194–195, New York, NY, USA. ACM.

[Sakai et al., 2005] Sakai, T., Terada, K., and Araragi, T. (2005). Robust online reputation mechanism by stochastic approximation. In Kudenko, D., Kazakov, D., and Alonso, E., editors, *Adaptive Agents and Multi-Agent Systems*, volume 3394 of *Lecture Notes in Computer Science*, pages 230–244. Springer.

[Schillo et al., 2000] Schillo, M., Funk, P., and Rovatsos, M. (2000). Using trust for detecting deceitful agents in artificial societies. *Applied Artificial Intelligence*, 14(8):825–848.

[Schimpf, 2002] Schimpf, J. (2002). Logical loops. In *ICLP'02*, pages 224–238.

[Seredynski et al., 2003] Seredynski, F., Zomaya, A., and Bouvry, P. (2003). Function optimization with coevolutionary algorithms. In *International Intelligent Information Processing and Web Mining Conference, June 2003, (Zakopane, Poland)*.

[Serrano et al., 2012] Serrano, E., Rovatsos, M., and Botia, J. (2012). A qualitative reputation system for multiagent systems with protocol-based communication. In

Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1, AAMAS '12, pages 307–314, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

[Stylios and Georgopoulos, 2008] Stylios, C. D. and Georgopoulos, V. C. (2008). Genetic algorithm enhanced fuzzy cognitive maps for medical diagnosis. In *FUZZ-IEEE*, pages 2123–2128. IEEE.

[Sutcliffe and Wang, 2012] Sutcliffe, A. and Wang, D. (2012). Computational modelling of trust and social relationships. *Journal of Artificial Societies and Social Simulation*, 15(1):3.

[Takashima et al., 2003] Takashima, E., Murata, Y., Shibata, N., and Ito, M. (2003). Self adaptive island GA. In *2003 Congress on Evolutionary Computation*, pages 1072–1079.

[Tan and Bhanu, 2006] Tan, X. and Bhanu, B. (2006). Fingerprint matching by genetic algorithms. *Pattern Recogn.*, 39(3):465–477.

[Tanese, 1989] Tanese, R. (1989). Distributed genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pages 434–439, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Teacy et al., 2006] Teacy, W. T. L., Patel, J., Jennings, N. R., and Luck, M. (2006). Travos: Trust and reputation in the context of inaccurate information sources. *Autonomous Agents and Multi-Agent Systems*, 12(2):183–198.

[Tuson, 1995] Tuson, A. L. (1995). Adapting operator probabilities in genetic algorithms. Technical report, Master's thesis, Evolutionary Computation Group, Dept. of Artificial Intelligence, Edinburgh University.

[Venketesh and Venkatesan, 2009] Venketesh, P. and Venkatesan, R. (2009). A Survey on Applications of Neural Networks and Evolutionary Techniques in Web Caching. *IETE Technical Review*, 26(3):171–180.

[Wang, 2006] Wang, H.-W. (2006). Portfolio selection with fuzzy mcdm using genetic algorithm: application of financial engineering. In *Proceedings of the 17th IASTED international conference on Modelling and simulation*, pages 597–602, Anaheim, CA, USA. ACTA Press.

[Wang et al., 2011] Wang, Y., Hang, C.-W., and Singh, M. P. (2011). A probabilistic approach for maintaining trust based on evidence. *J. Artif. Int. Res.*, 40(1):221–267.

[Wong et al., 2010] Wong, K.-C., Leung, K.-S., and Wong, M.-H. (2010). Protein structure prediction on a lattice model via multimodal optimization techniques. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 155–162, New York, NY, USA. ACM.

[Xin et al., 2012] Xin, N., Gu, X., Wu, H., Hu, Y., and Yang, Z. (2012). Application of genetic algorithm-support vector regression (ga-svr) for quantitative analysis of herbal medicines. *Journal of Chemometrics*, 26(7):353–360.

[Yu and Singh, 2002] Yu, B. and Singh, M. P. (2002). An evidential model of distributed reputation management. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, AAMAS '02, pages 294–301, New York, NY, USA. ACM.

[Yuan, 2005] Yuan, B. (2005). A hybrid approach to parameter tuning in genetic algorithms. In *In IEEE International Conference on Evolutionary Computation*.

[Yun and Gen, 2003] Yun, Y. and Gen, M. (2003). Performance analysis of adaptive genetic algorithms with fuzzy logic and heuristics. *Fuzzy Optimization and Decision Making*, 2:161–175. 10.1023/A:1023499201829.

[Zacharia and Maes, 2000] Zacharia, G. and Maes, P. (2000). Trust management through reputation mechanisms. *Applied Artificial Intelligence*, 14(9):881–907.

[Zhang et al., 2007] Zhang, J., hung Chung, H. S., Member, S., and lun Lo, W. (2007). Clustering-based adaptive crossover and mutation. *IEEE Trans. on Evolutionary Computation*, pages 326–335.

[Zhang and Szeto, 2005] Zhang, J. and Szeto, K. Y. (2005). Mutation matrix in evolutionary computation: an application to resource allocation problem. In *Proceedings of the First international conference on Advances in Natural Computation - Volume Part III*, ICNC'05, pages 112–119, Berlin, Heidelberg. Springer-Verlag.

[Zhao et al., 2008] Zhao, X., Lee, M.-E., and Kim, S.-H. (2008). Improved image segmentation method based on optimized threshold using genetic algorithm. In

Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications, AICCSA '08, pages 921–922, Washington, DC, USA. IEEE Computer Society.